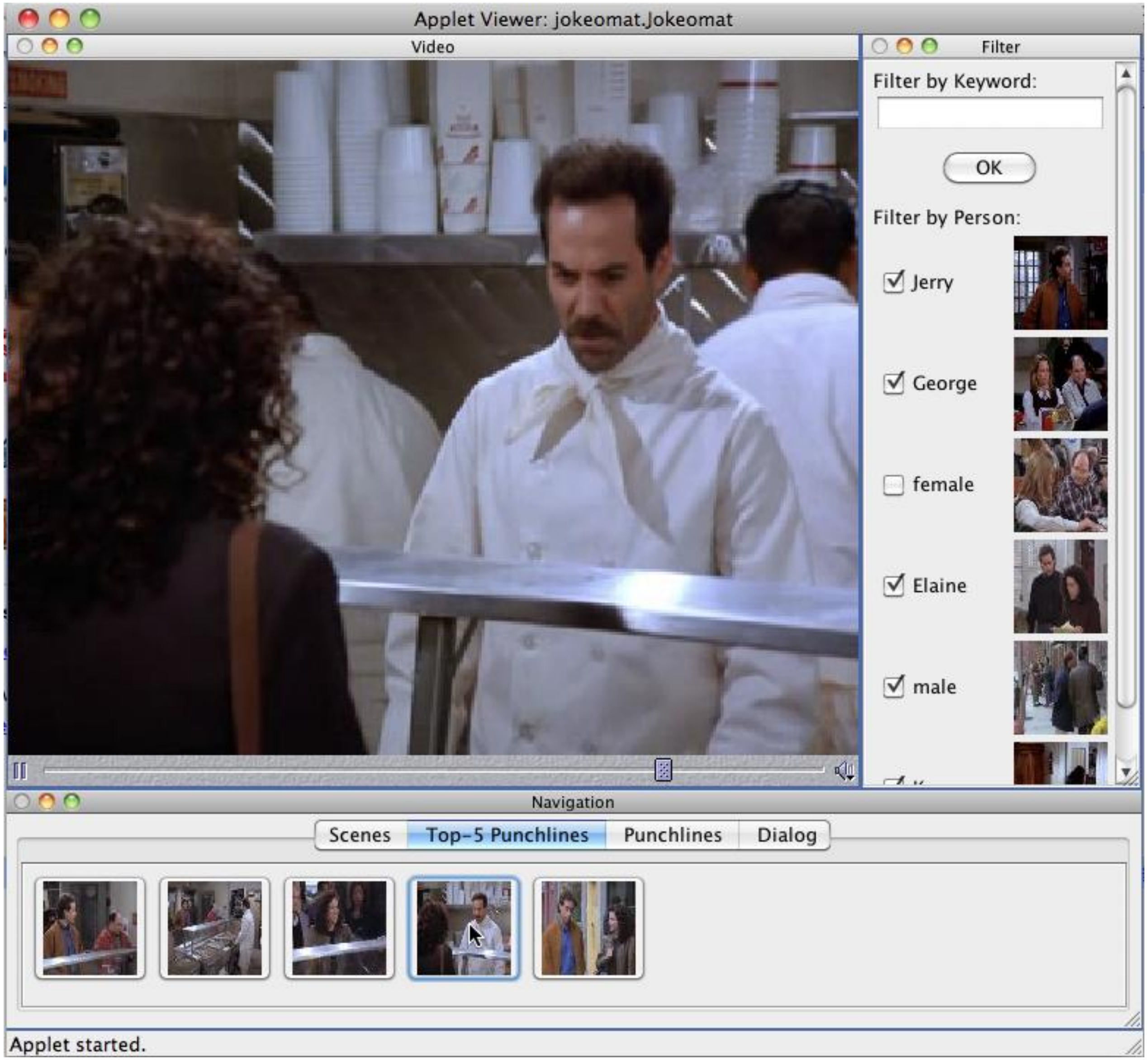
Productive GMM Training with SEJITS for Speaker Diarization

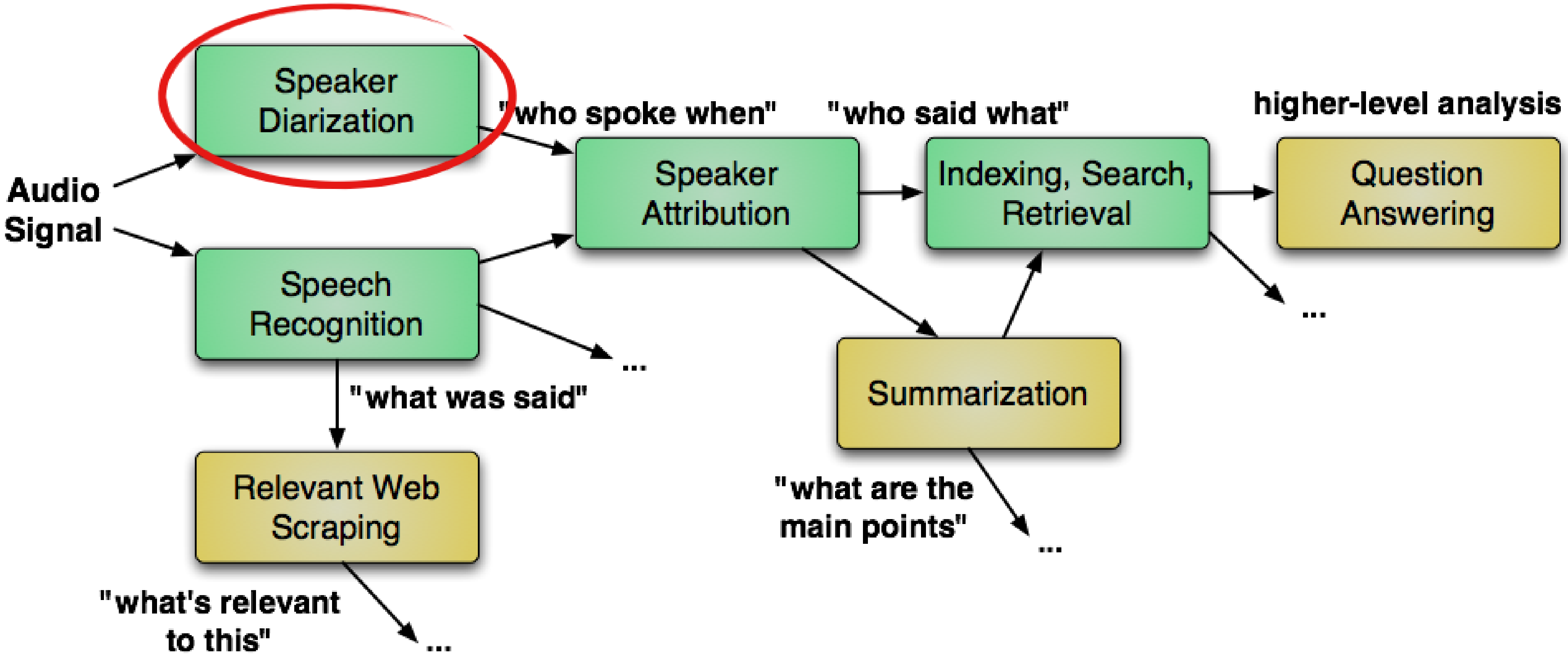
Katya Gonina, Henry Cook, Shoiab Kamil,
Gerald Friedland, Armando Fox, David Patterson

ParLab Retreat, June 2, 2011

The Meeting Diarist



Components of the Meeting Diarist

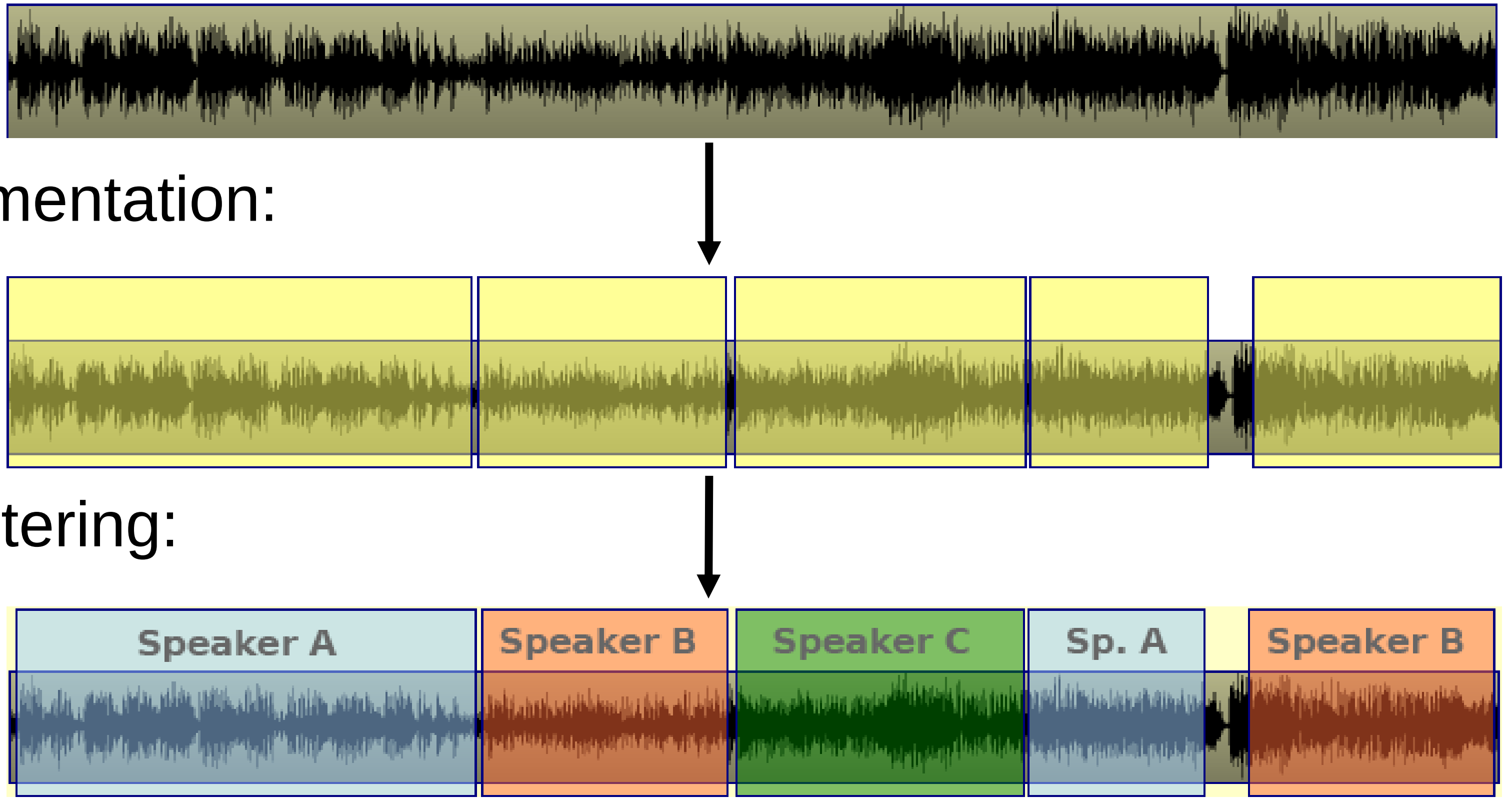


Speaker Diarization

Audio track:

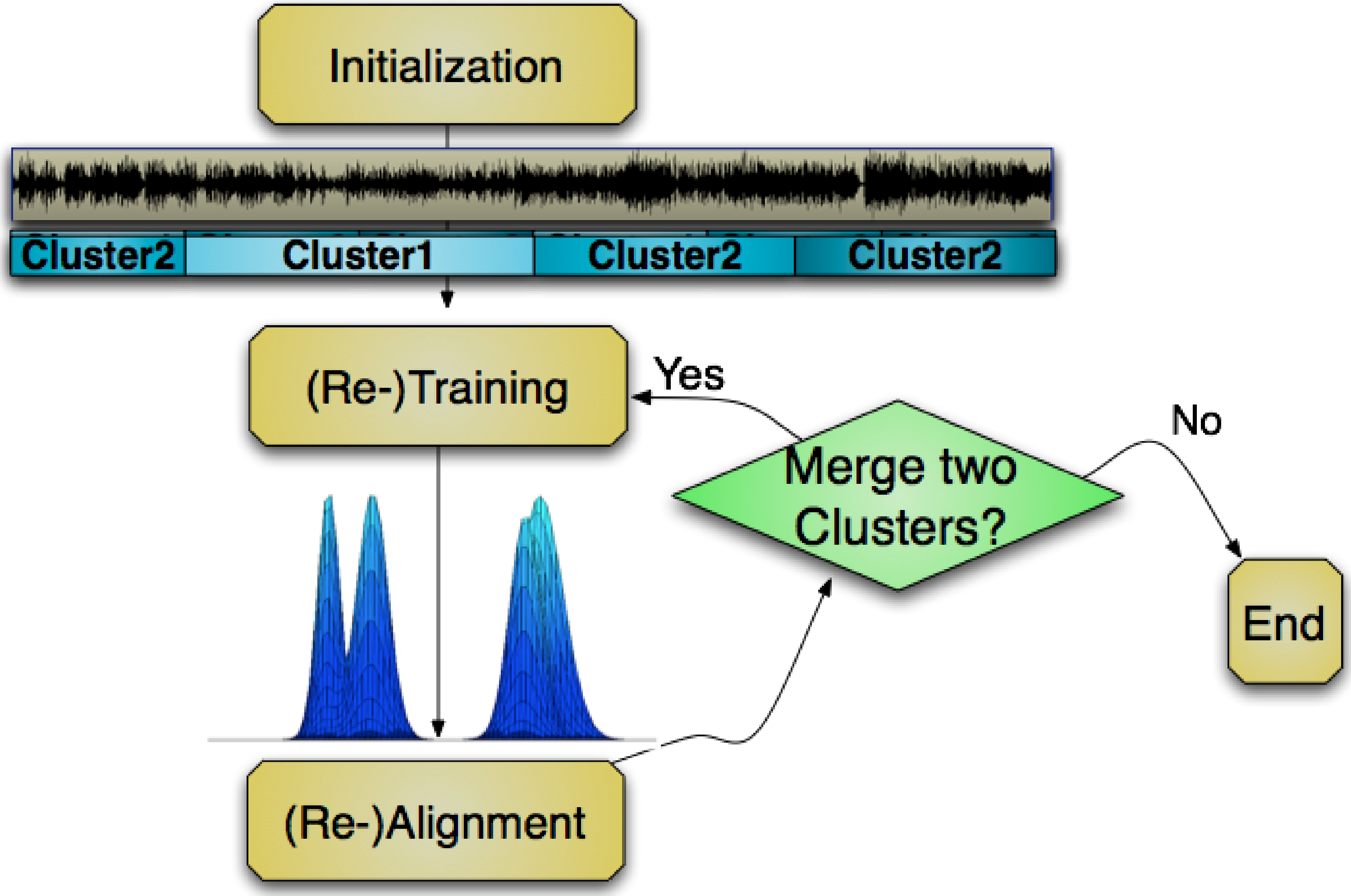
Segmentation:

Clustering:



Estimate “who spoke when” with no prior knowledge of speakers, #of speakers, words, or language spoken.

Speaker Diarization: Core Algorithm



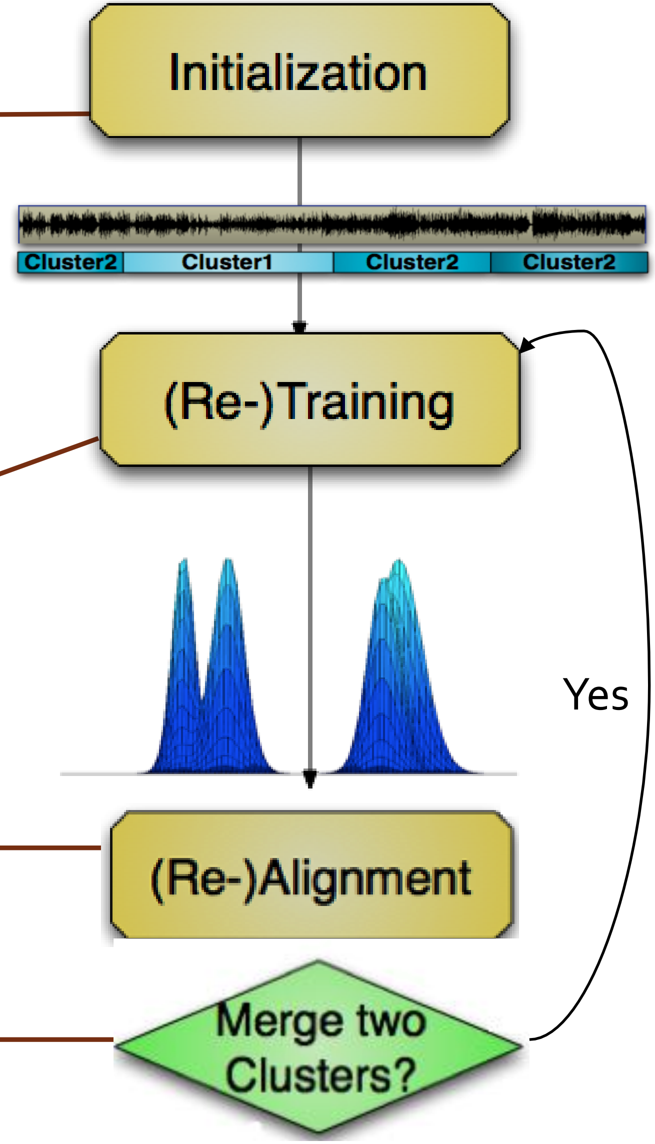
Parallelization of Diarization

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

[1] H. Cook, E. Gonina, S. Kamil, G. Friedland, D. Patterson, A. Fox. CUDA-level Performance with Python-level Productivity for Gaussian Mixture Model Applications. USENIX HotPar Workshop, 2011.

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)

        # 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 gmm1idx in range(len(iter_bic_list)):
            for gmm2idx in range(gmm1idx+1, len(iter_bic_list)):
                g1, d1 = iter_bic_list[gmm1idx]
                g2, d2 = iter_bic_list[gmm2idx]
                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)
```



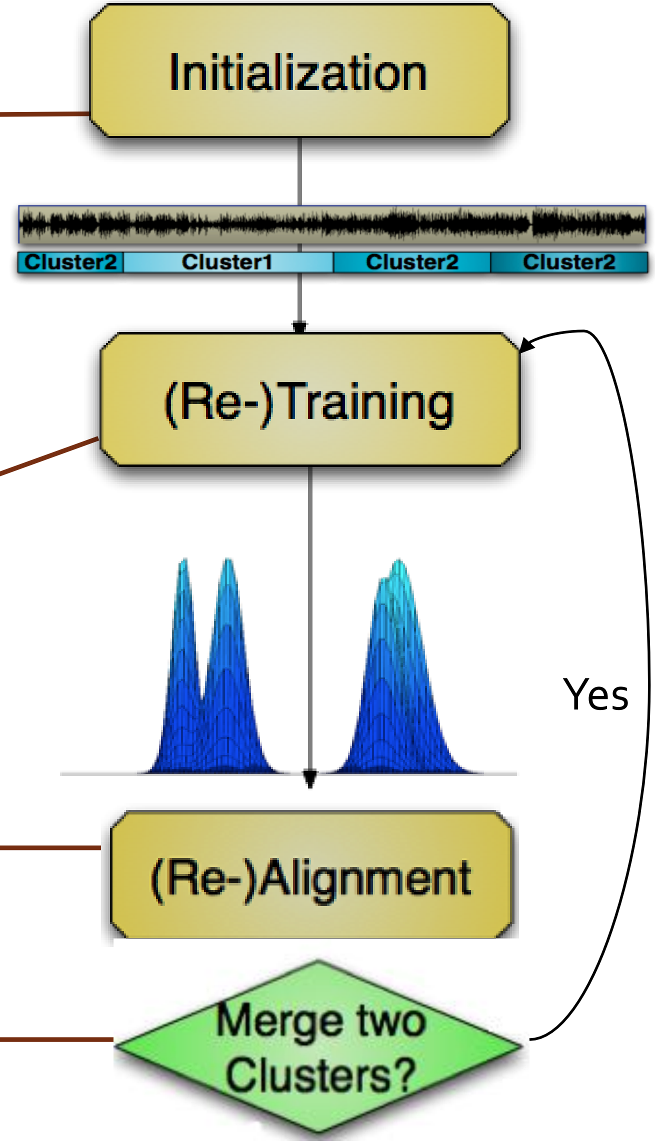
Speaker Diarization in Python

```
def AHC(self):

    # Ge                                          rs and train each GMM on a cluster
    per_
    init                                          nge(per_cluster, self.N, per_cluster)))
    for

    # 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_                            _on_votes(most_likely, self.X)

        for g,

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

        for gmm1idx in range(len(iter_bic_list)):
            for gmm2idx in range(gmm1idx+1, len(iter_bic_list)):
                g1, d1 = iter_bic_list[gmm1idx]
                g2, d2 = iter_bic_list[gmm2idx]
                score = 0.0
                new_gmm, score = com                  .concatenate((d1, d2)))
                if score > best_BIC_
                    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)
```

new_gmm_list(M,D)
g.train(x)

g.train(x)

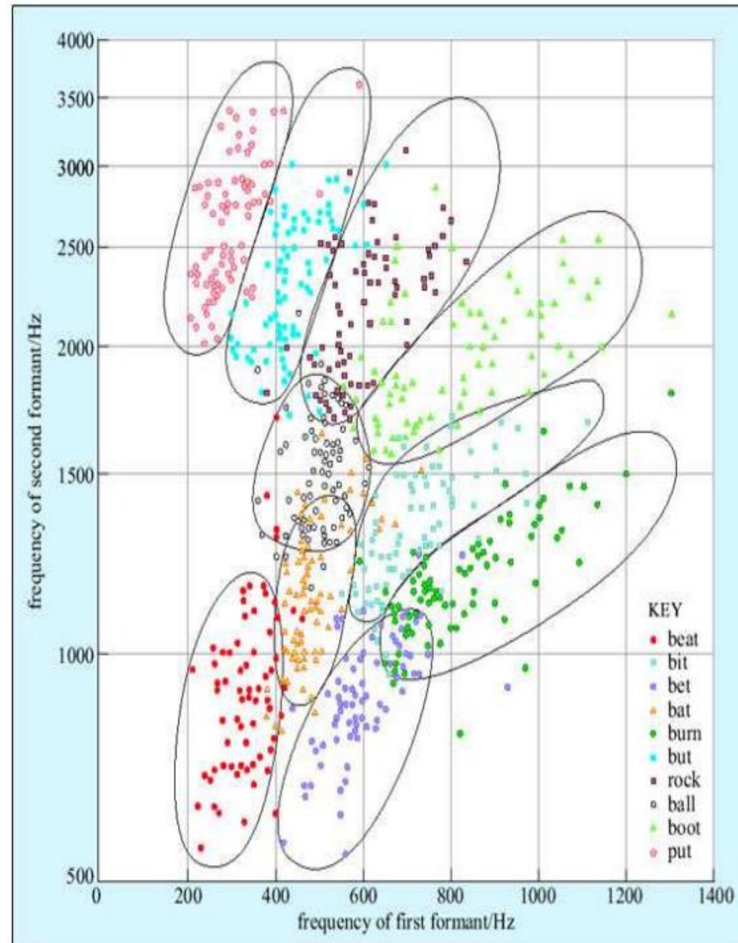
g.train(x)



Gaussian Mixture Models & Training

Clustering with Gaussian Mixture Models

- 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 (*n*) and a covariance (*s*) parameters
  - Gaussians in the mixture are weighted with weight *p*



Example GMM in two dimensions
(Source: Dan Klein, UCB)

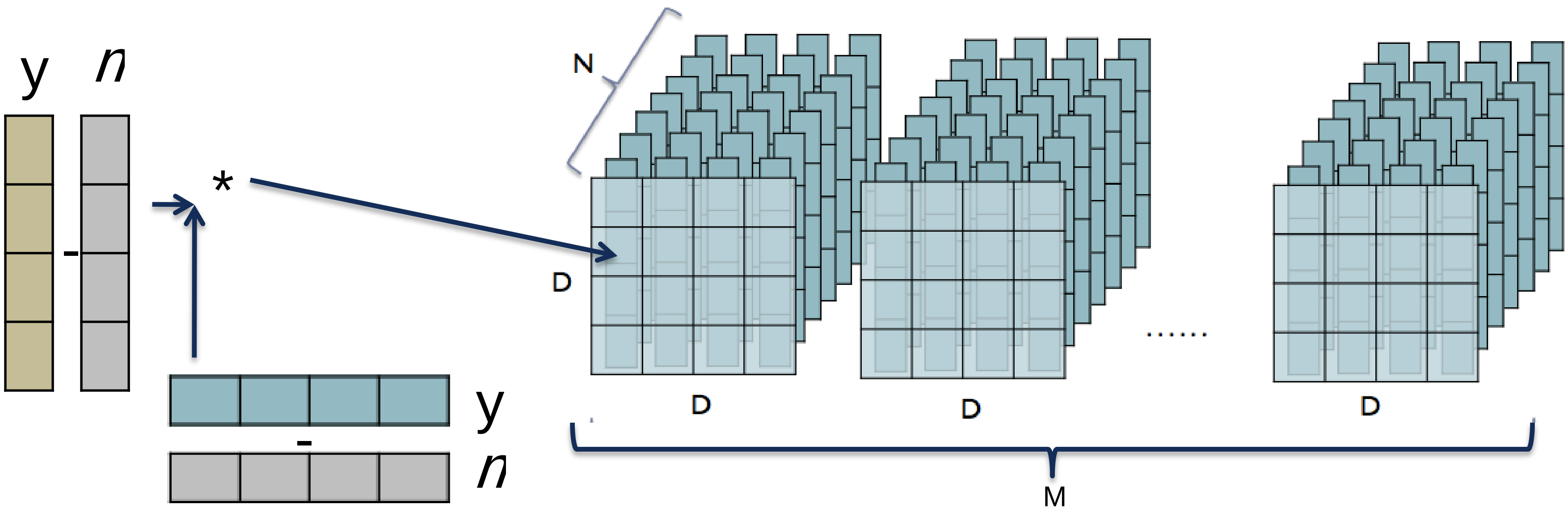
GMM Training using EM Algorithm

- Given a set of observations/events – find the maximum likelihood estimates of the set of Gaussian Mixture parameters (n, s, p) 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

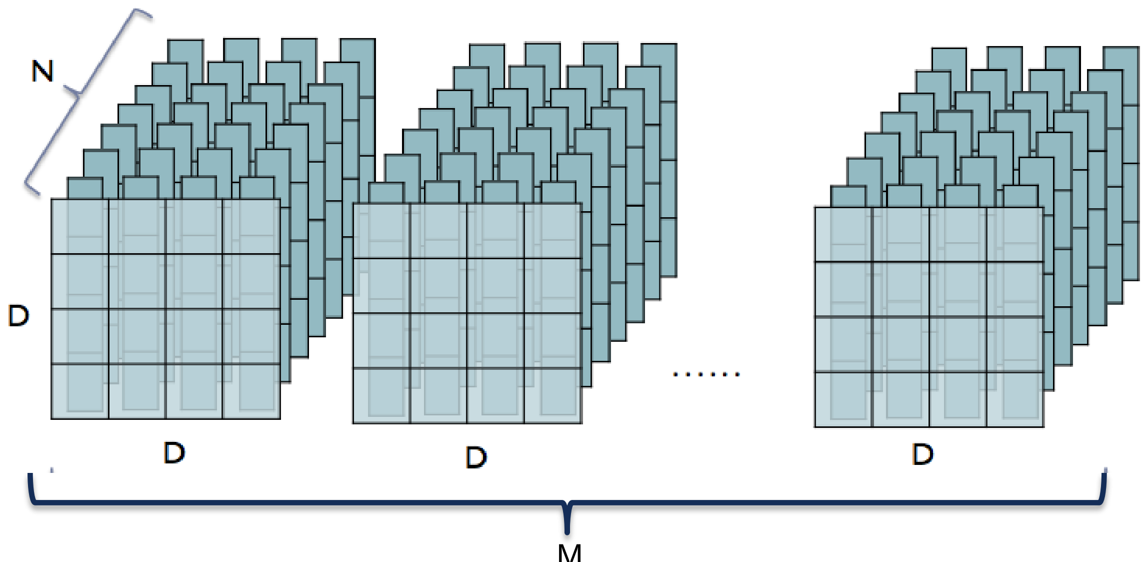
Covariance Matrix Computation

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

$$\bar{R}_k = \frac{1}{\bar{N}_k}\sum_{n=1}^{N}(y_n - \bar{\mu}_k)(y_n - \bar{\mu}_k)^t p_{x_n|y_n}(k|y_n, \theta^{(i)})$$

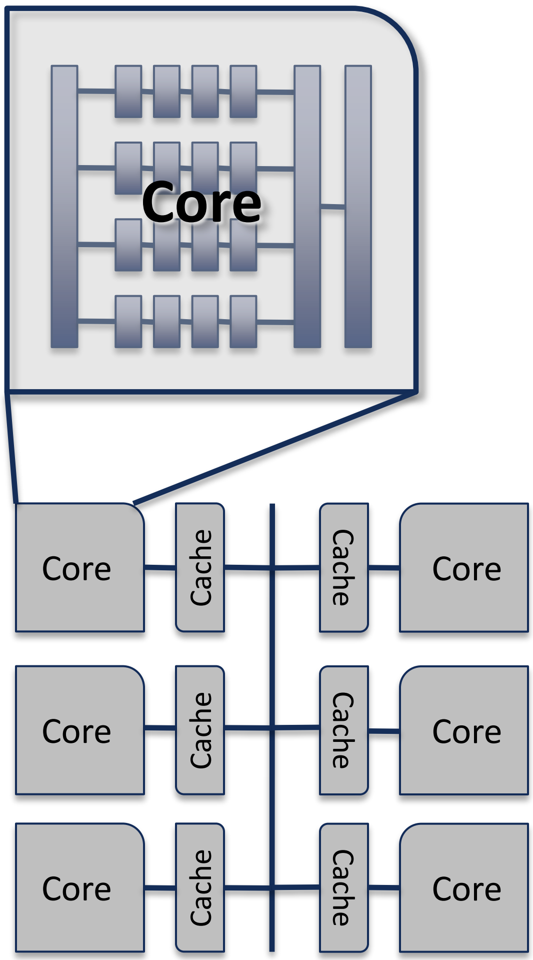


Covariance Matrix Computation



- 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 covar matrix (N)
- -> **Multiple code variants** to perform the same computation in different ways

Manycore Parallel Platform

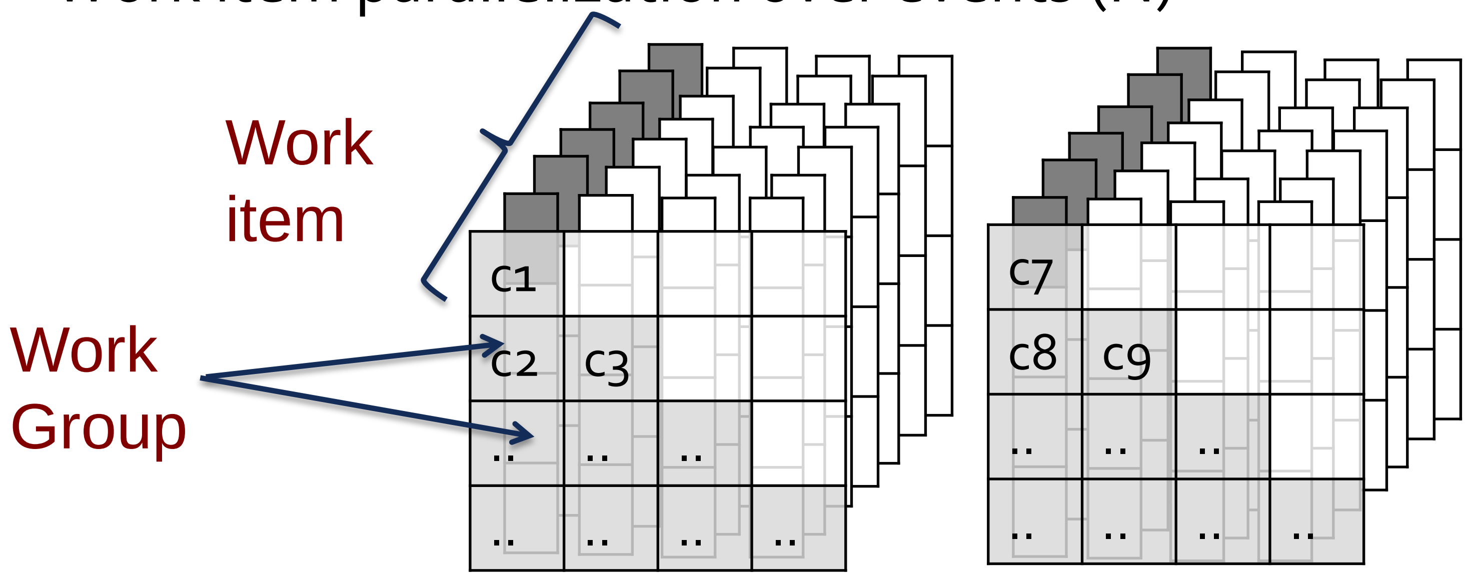


- Two levels of parallelism:
  - Work-groups – parallelized across cores (CUDA threadBlock)
  - Work-groups' work-items – executed on a single core, utilizing within-core parallelism (CUDA thread)
- Per-core local memory

Code Variants

Code Variants - Example

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



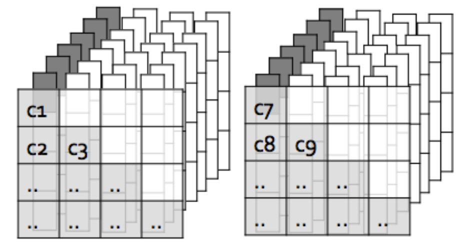
[2]. A. D. Pangborn. Scalable data clustering using gpus. Master's thesis, Rochester Institute of Technology, 2010.

Covariance Matrix Computation – Code Variants

for each component m in M comps
  for each cell c in DxD/2 cells → Work group

for each event n in N events → Work item

V1 add nth contribution to c of m → Seq.




```
for each cell c in DxD/2 cells          → Work group
  for each event n in N events          → Work item
    for each component m in M comps
      add nth contribution to c of m    → Seq
```

V2

.....

Covariance Matrix Computation – Code Variants Summary

V1

```
for each component m in M comps
  for each cell c in DxD/2 cells        → Work group
    for each event n in N events        → Work item
      add nth contribution to c of m    → Seq
```

V2

```
for each cell c in DxD/2 cells          → Work group
  for each event n in N events          → Work item
    for each component m in M comps
      add nth contribution to c of m    → Seq
```

V3

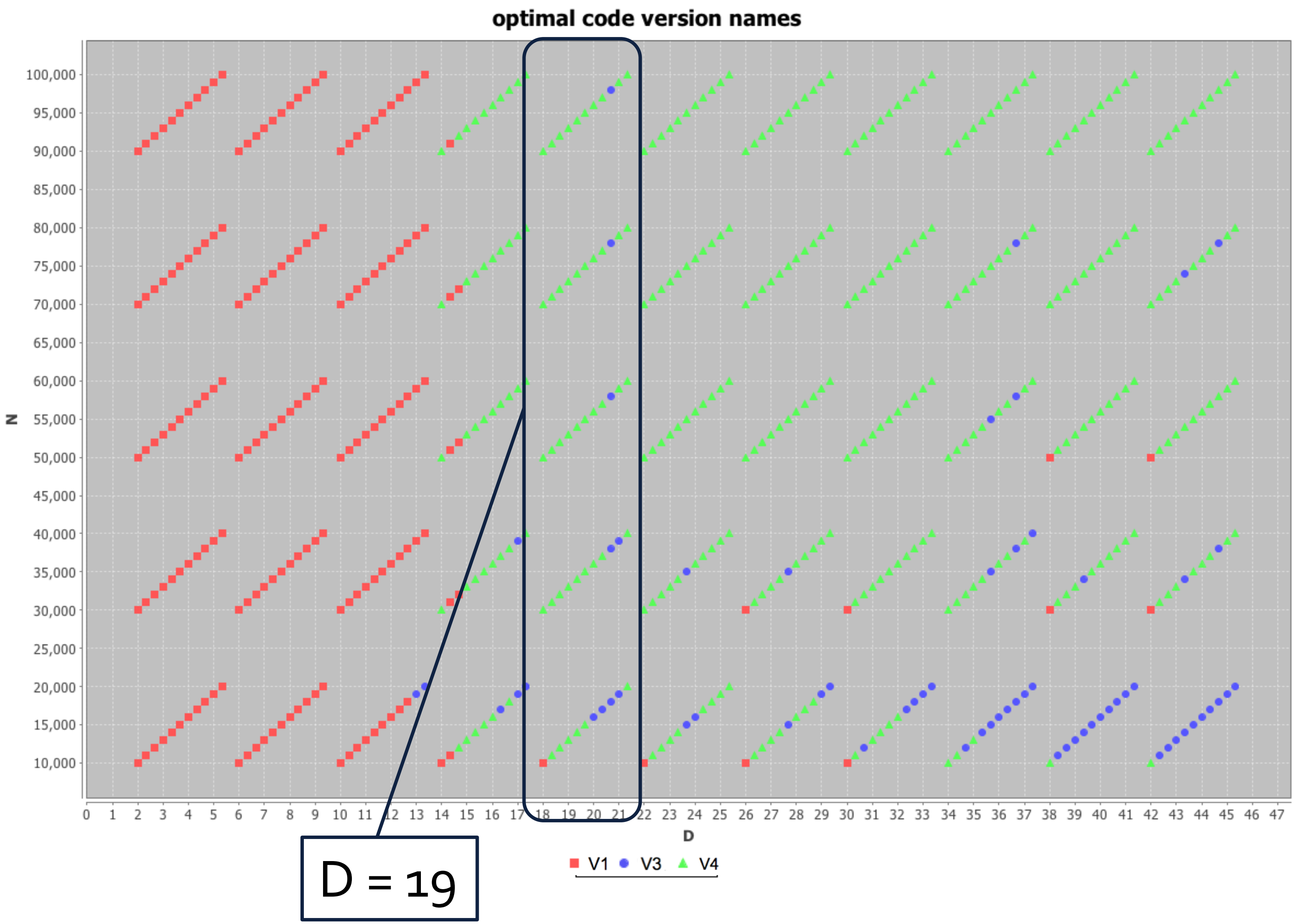
```
for each component m in M comps         → Work group
  for each cell c in DxD/2 cells        → Work item
    for each event n in N events
      add nth contribution to c of m    → Seq
```

V4

```
for each block b in B event blocks
  for each component m in M comps       → Work group
    for each cell c in DxD/2 cells      → Work item
      for each event n in N/B events
        add nth contribution to c of m  → Seq

for each component m in M comps
  for each block b in B event blocks
    sum partial contributions to m from b → Seq
```

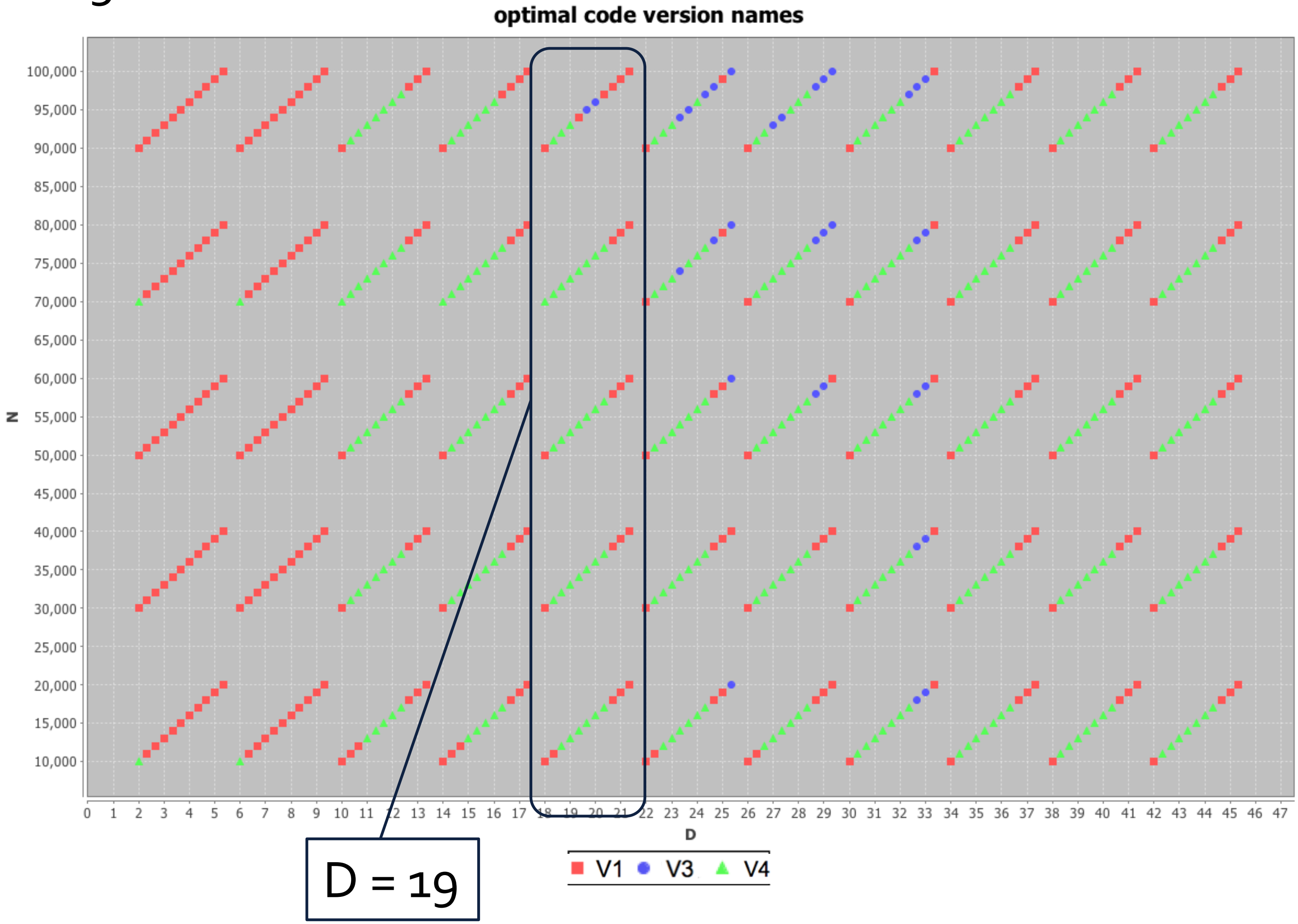
Results – Code Variant Performance

GTX480



Results – Code Variant Performance

GTX285



Results - Code Variant Selection

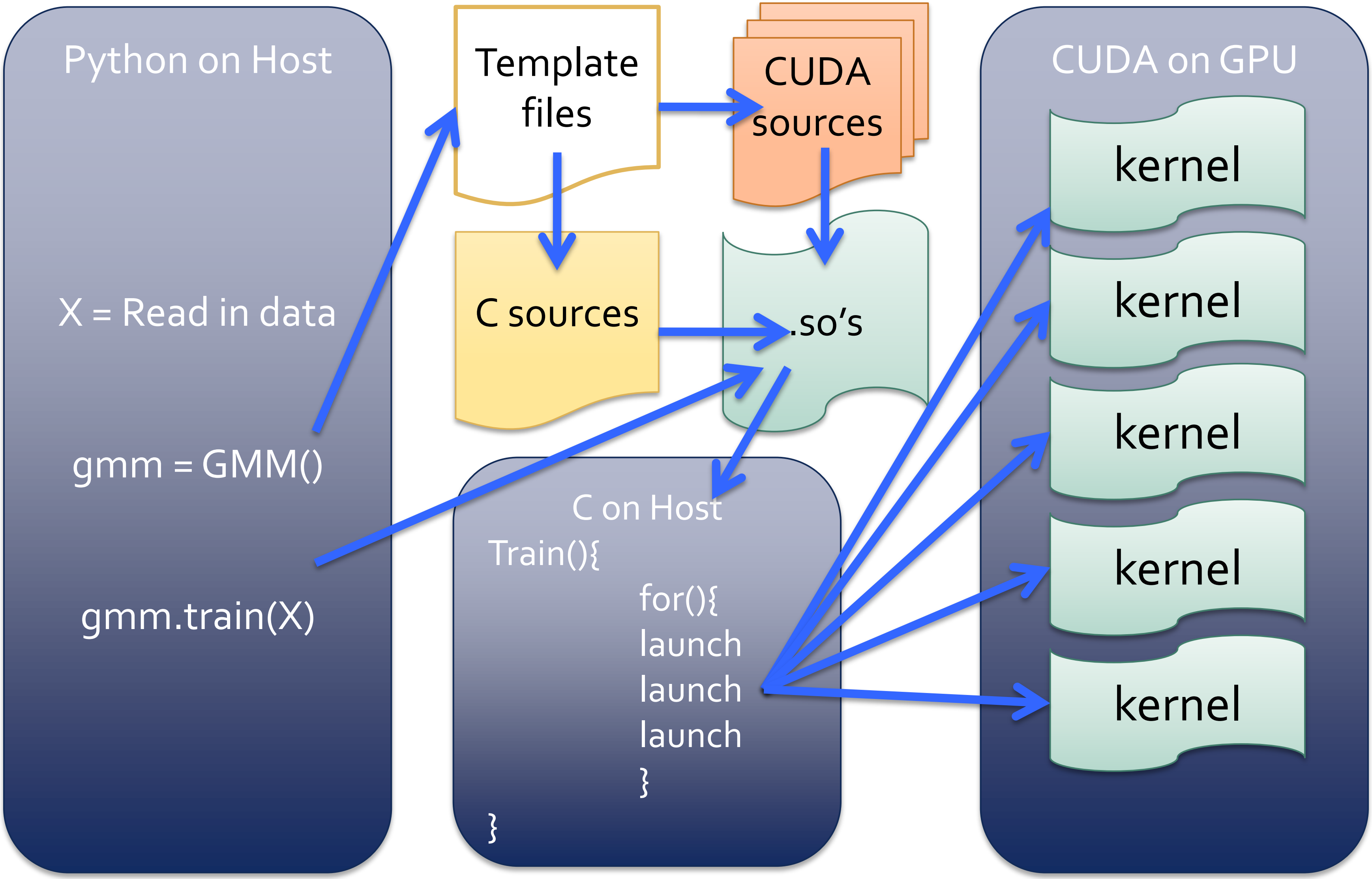
- Using best-performing code variant gave 32% average improvement in matrix computation time compared to always using original hand-coded variant (D: 1 to 36, M: 1 to 128, N: 10K to 150K)
- Performance gap increases with larger problem sizes (75.6% for D=36, M=128, N=500,000)

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

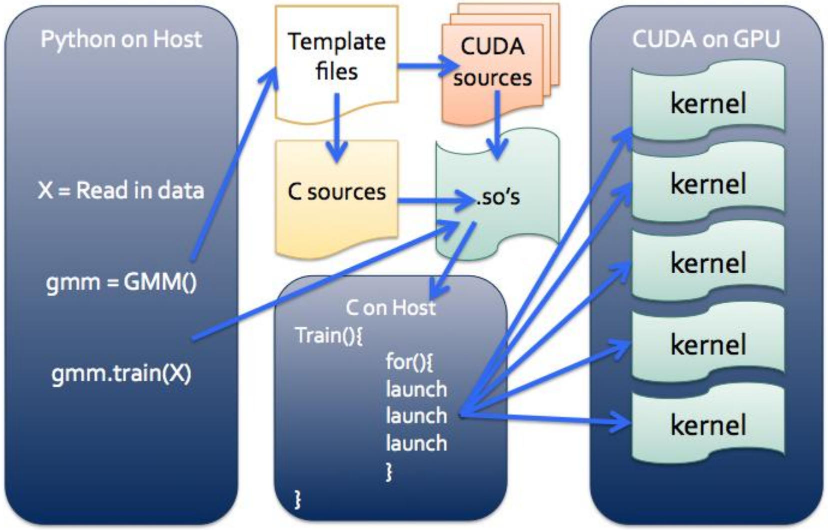
SEJITS Framework: Overview



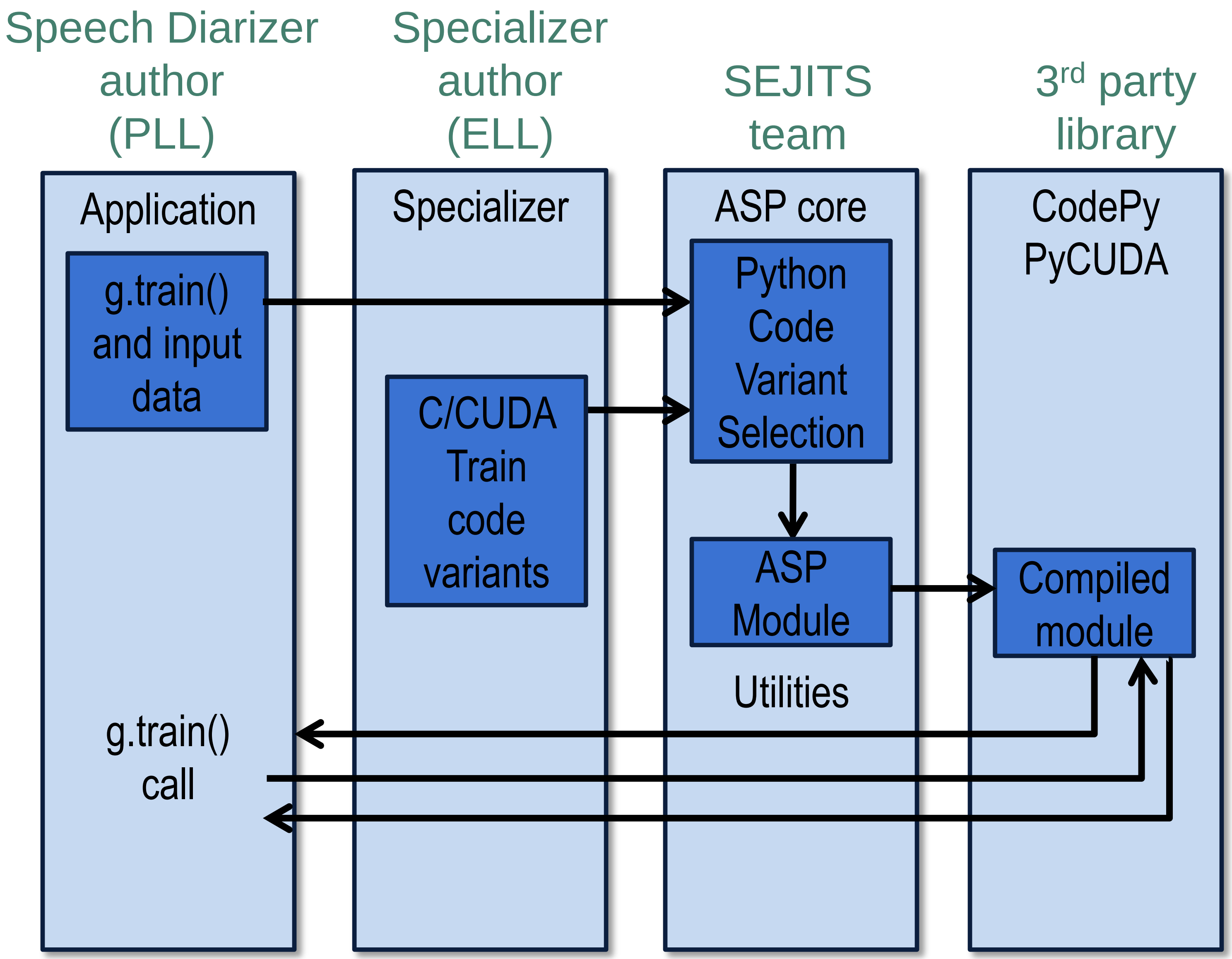
SEJITS Framework

- 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



Separation of Concerns



Speaker Diarization in Python

Python

```
def AHC(self):

    # Get th                                                  ach GMM on a cluster
    per_clus
    init_tra                                                , self.N, per_cluster)))
    for g, x
        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
        spli                                   votes(most_likely, self.X)

        for

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

        for gmm1idx in range(len(iter_bic_list)):
            for gmm2idx in range(gmm1idx+1, len(iter_bic_list)):
                g1, d1 = iter_bic_list[gmm1idx]
                g2, d2 = iter_bic_list[gmm2idx]
                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)
```

new_gmm_list(M,D)

g.train(x)

C

.....

Speaker Diarization in Python

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)

        # 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
        most_likely = likelihoods.argmax(axis=1)

        # Across 2.5 secs of observations, vote on which c
        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 gmm1idx in range(len(iter_bic_list)):
            for gmm2idx in range(gmm1idx+1, len(iter_bic_list)):
                g1, d1 = iter_bic_list[gmm1idx]
                g2, d2 = iter_bic_list[gmm2idx]
                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)
```

C

15x LOC reduction

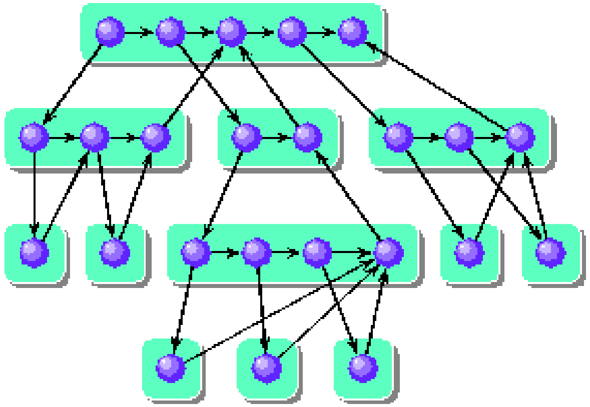
......

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

Cilk Backend

- 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/

Results – Portability & Maintenance

- 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 (200 x faster than realtime)
- Future work:
  - Further specialize train kernel
  - SEJITize other components
  - Improve code variant selection mechanism

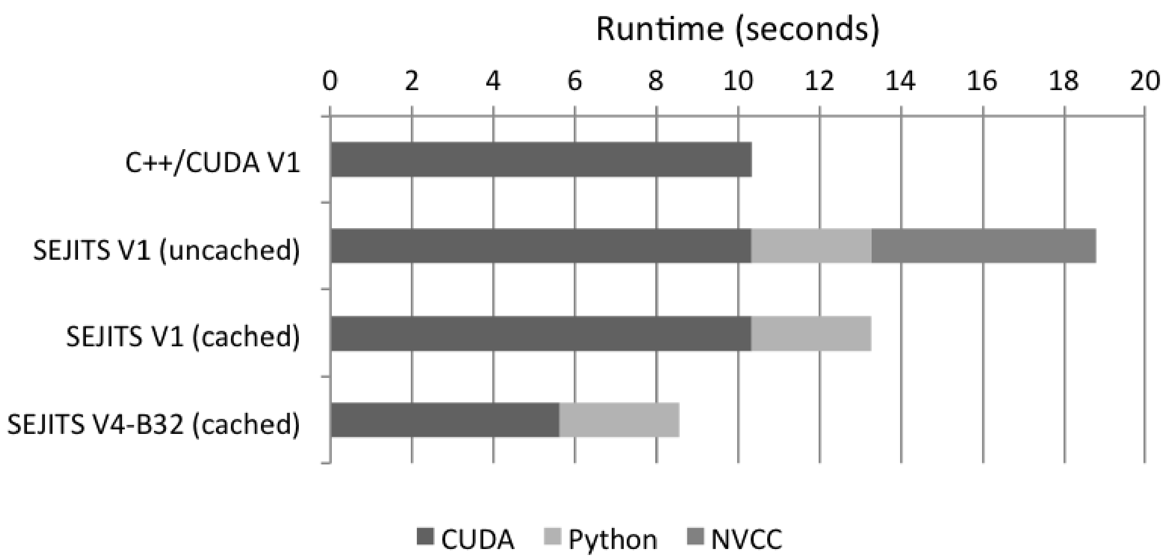
Thank you!

Questions?

Backup Slides

Results – Specializer Overhead in AHC

- Initial invocation – 81% overhead dude to complier invocations
- Future runs using automatically determined optimal code variant achieve 17% performance *improvement* over the original GPU implementation (V1)



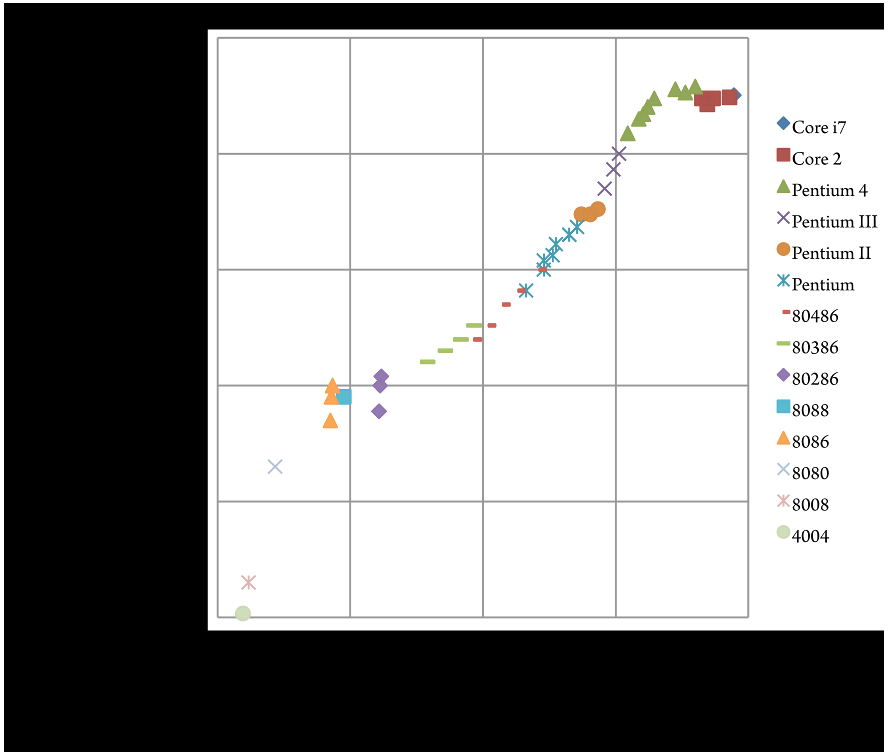
ASP vs Auto-tuning Libraries

	ATLAS	FFTW, Spiral, OSKI	ASP/GMM	ASP/Stencil	Delite/OptiML	Copperhead
Autuning of code	✔	✔	✔	✔	✔	✔
Based on runtime information		✔	✔	✔	✔	✔
Based on higher-order func				✔	✔	✔
Using reusable framework			✔	✔	✔	✔
Embedded in HLL			✔	✔	✔	✔

The shift to parallel processing

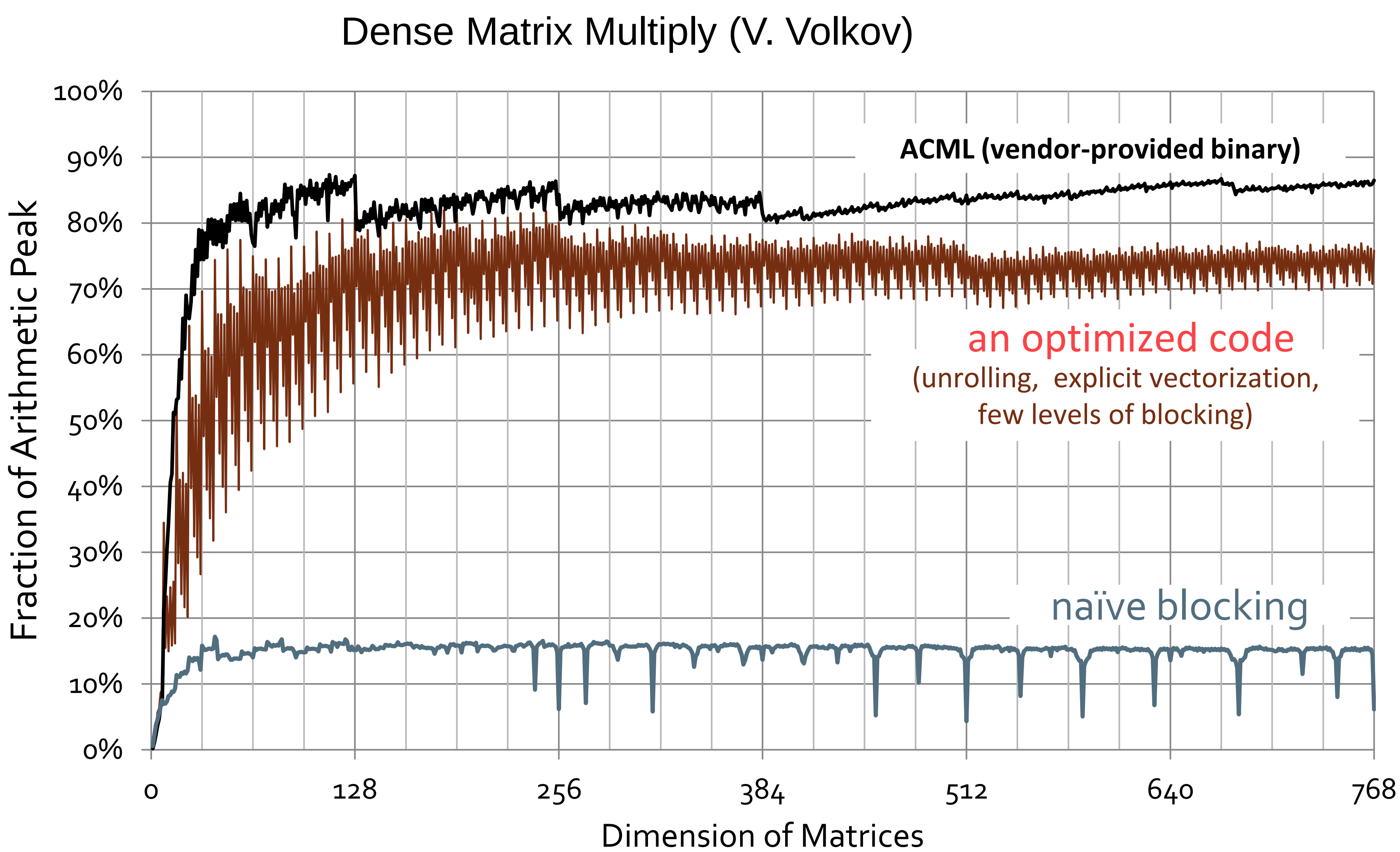
- Parallel processing is here

> “*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

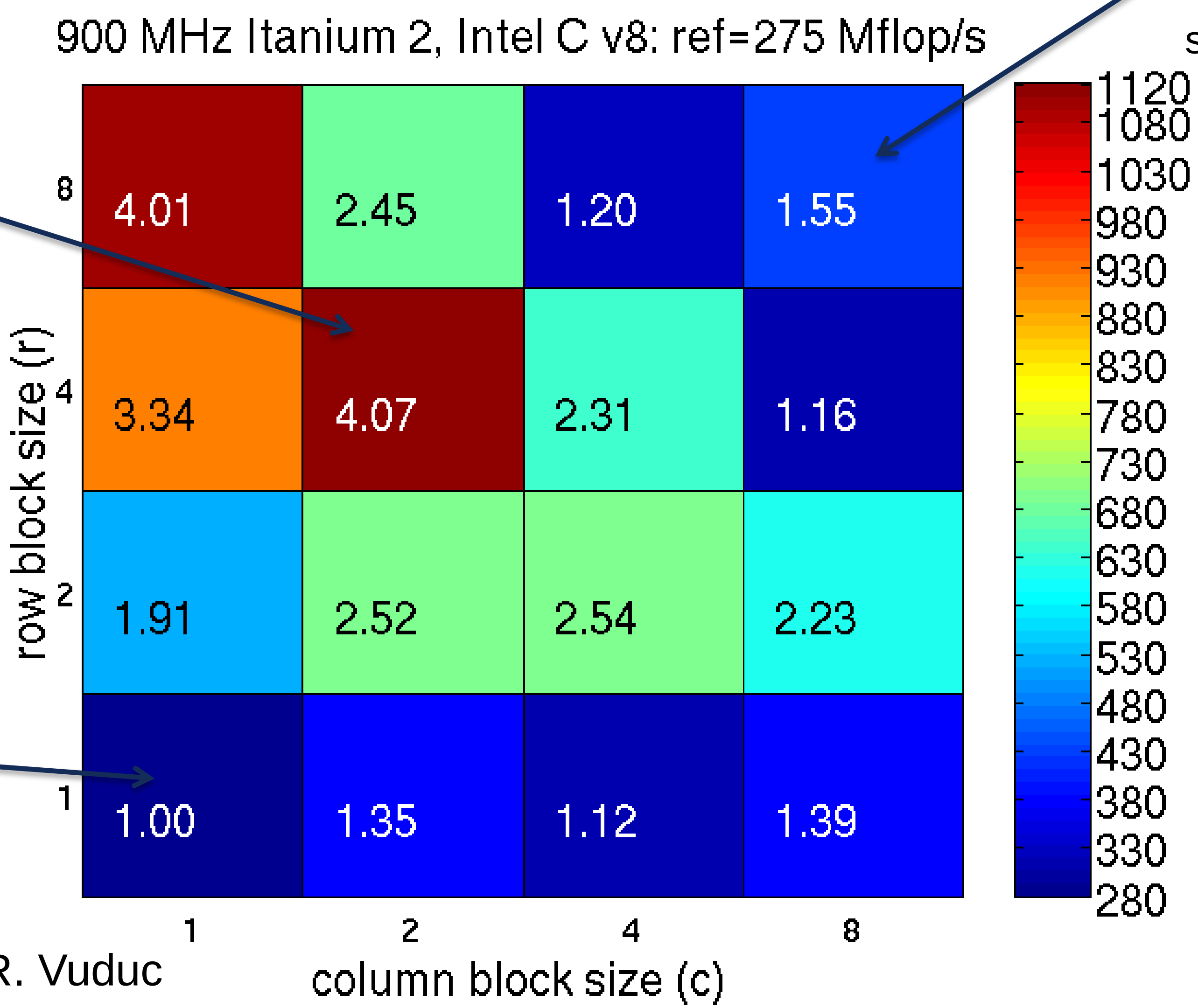
Writing Fast Code is Hard



Finding Best Implementation is Hard

Implementation Based on structure of data

Best performing



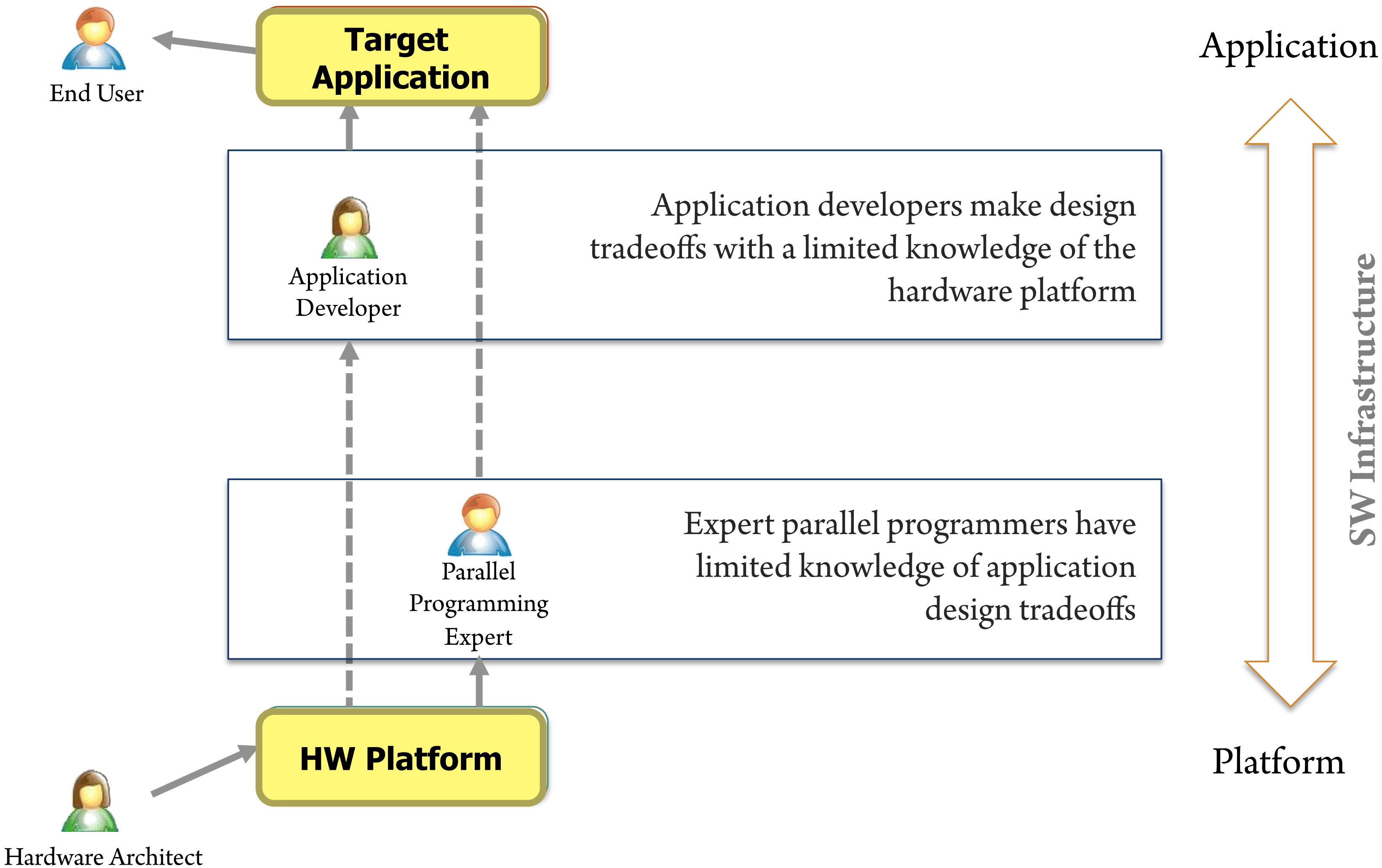
Naïve implementation

Figure from R. Vuduc

Productivity vs Performance

- 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



Outline

- SEJITS approach
- Gaussian Mixture Model & Applications
- Covariance Matrix Computation & Code Variants
- Specialization
- Results
- Conclusion & Future Work

Selective Embedded Just-In-Time Specialization (SEJITS)

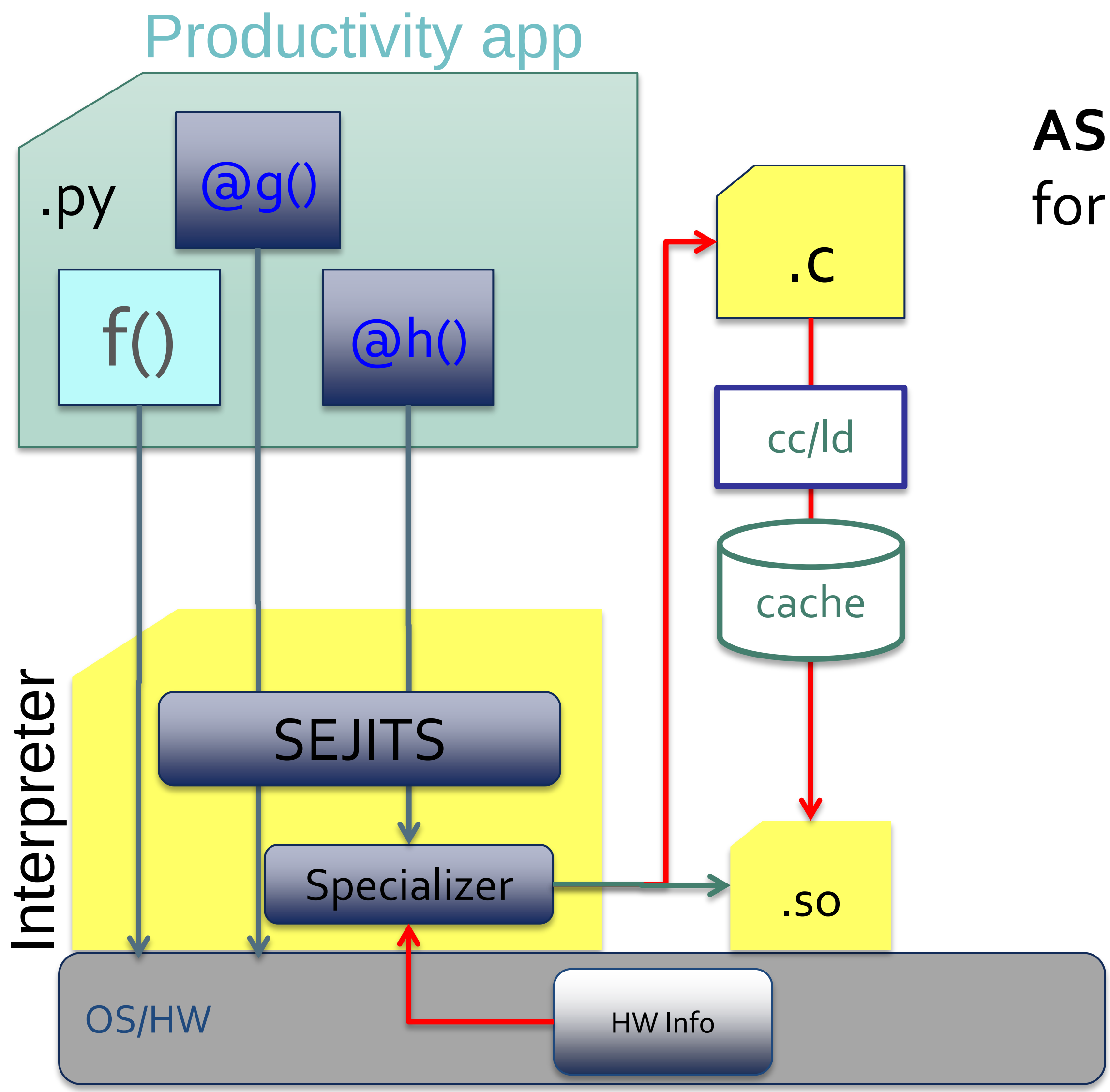
Key Idea: Generate, compile, and execute high performance parallel code at runtime using code transformation, introspection, variant selection and other features of high-level languages.

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

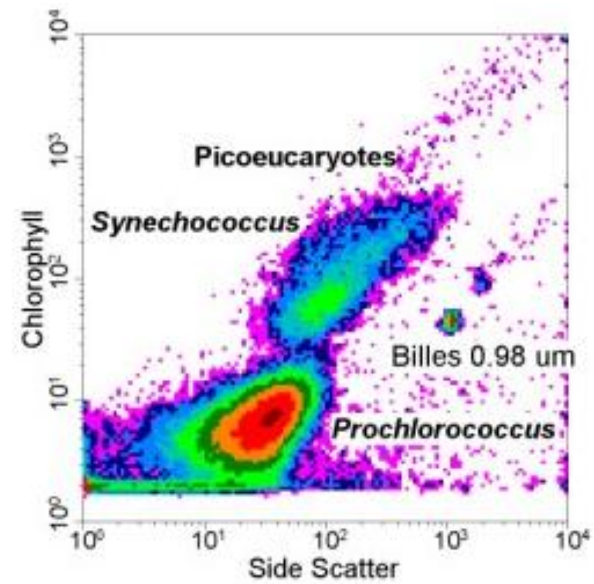
Selective Embedded JIT Specialization (SEJITS)



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…



Results – Specializer Overhead

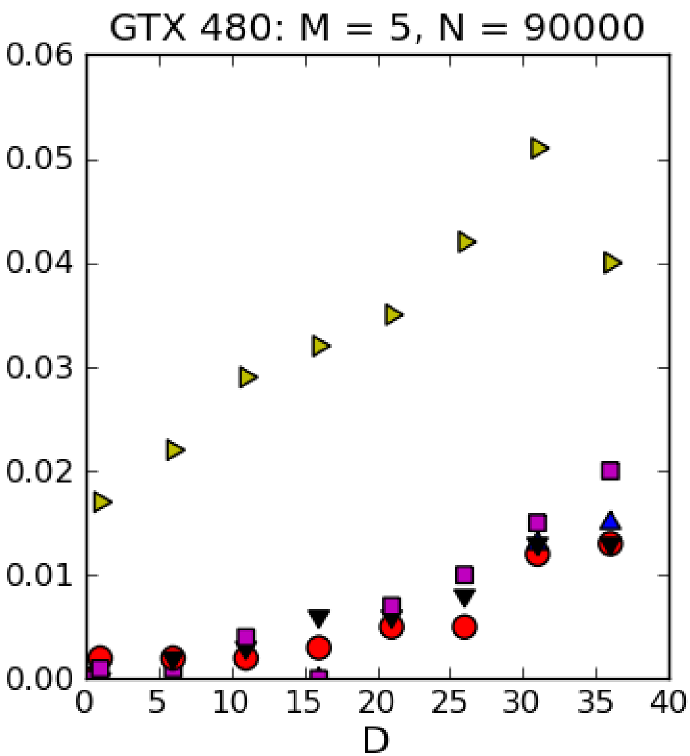
- Application example – Agglomerative Hierarchical Clustering for Speaker Diarization
  - Uses GMMs to represent distribution of audio features for speakers in a recorded meeting
  - Iteratively trains GMMs using different number of components each time and measuring which number of components best fits the data
  - Number of components in the best GMM corresponds to number of speakers in the meeting

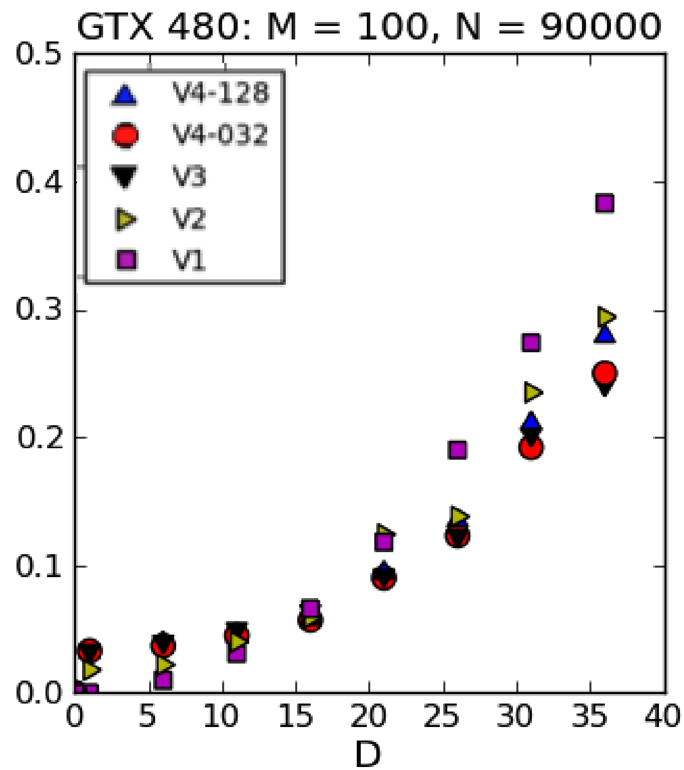
Conclusions & Future Work

- 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

Results – Version Comparison (Raw CUDA)

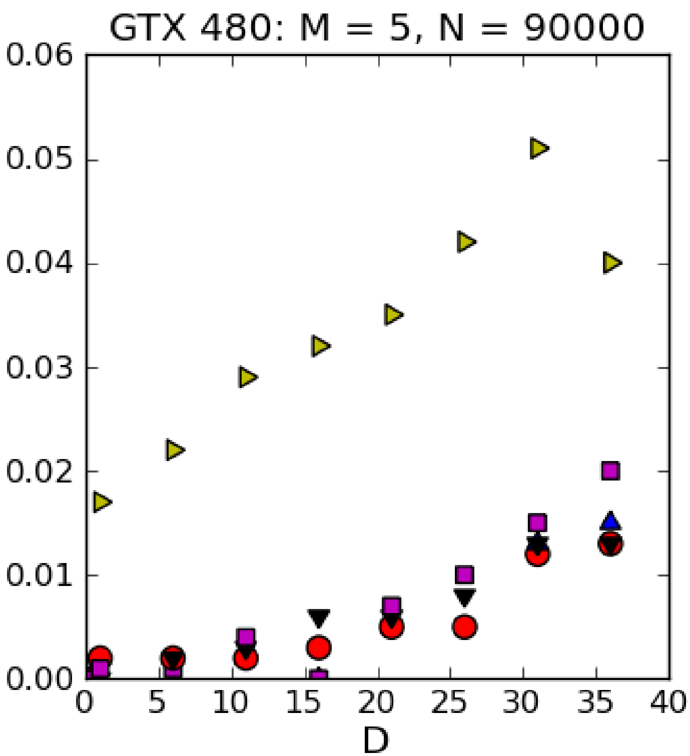
GTX480 – Varying D

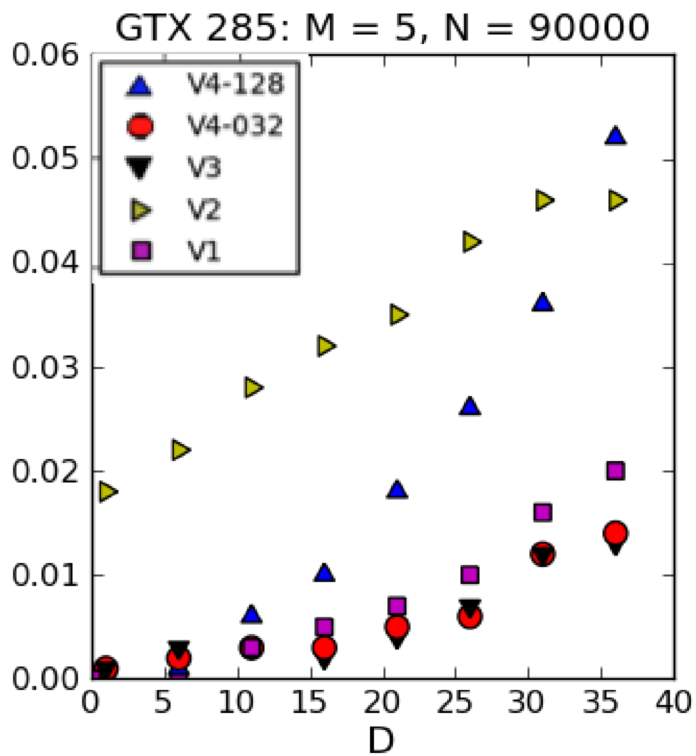




Results – Version Comparison (Raw CUDA)

GTX285 vs. 480





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

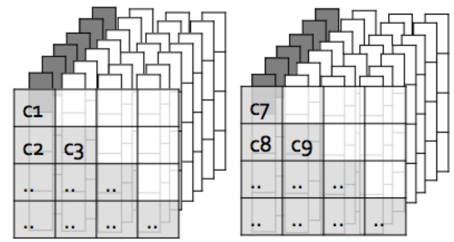
- Evaluation platforms:
  - GTX480 (Fermi)
    - 14 SM, 32 SIMD, 48K shared mem, 3GB DRAM
  - GTX 285
    - 30 SM, 8 SIMD, 16K shared mem, 1GB DRAM
  - CUDA SDK 3.2
  - NVCC 3.2

Covariance Matrix Computation – Code Variants

for each component m in M comps
  for each cell c in DxD/2 cells → Work group

for each event n in N events → Work item

V1 add nth contribution to c of m → Seq



for each cell c in DxD/2 cells → Work group

for each event n in N events → Work item

for each component m in M comps
V2 add nth contribution to c of m → Seq

.....