A Specialization Framework for Audio Content Analysis

Katya Gonina
with Henry Cook, Eric Battenberg, Gerald Friedland* and Kurt Keutzer

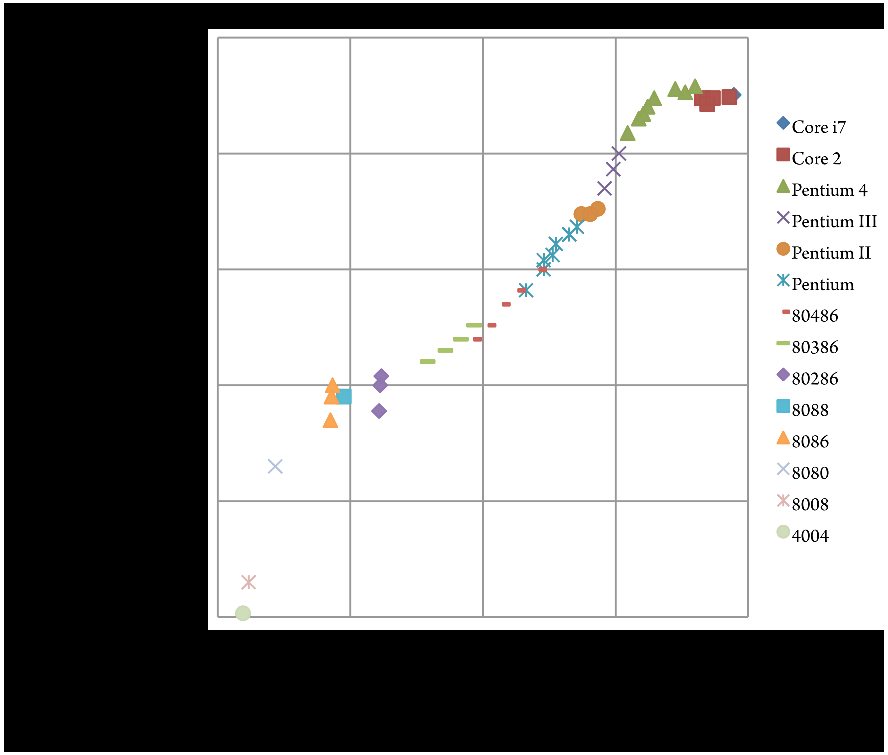
UC Berkeley ParLab, *International Computer Science Institute

January 18, 2012

The shift to parallel processing

- Parallel processing is here

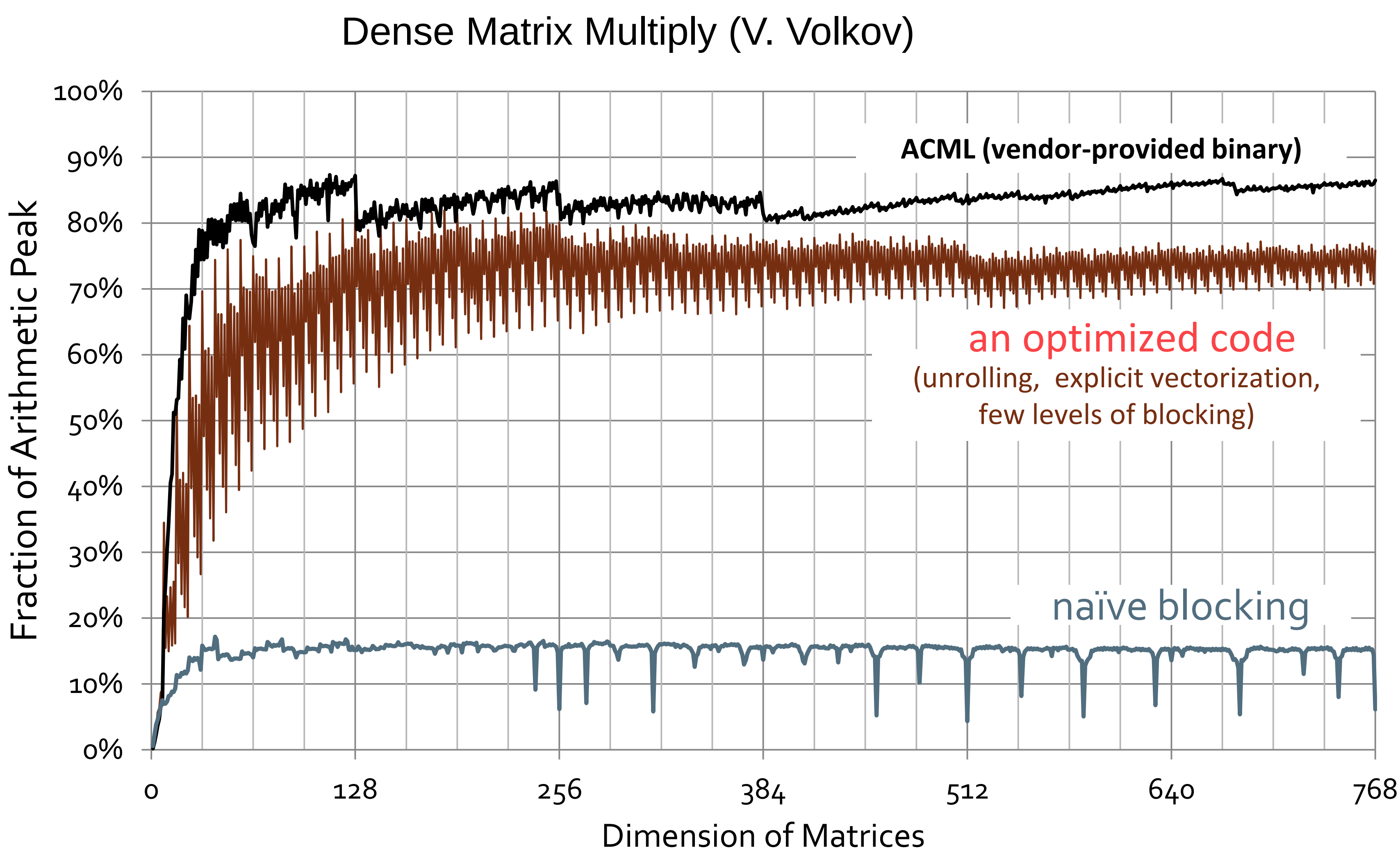
> *"This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures."*
>
> \- The Berkeley View [1]



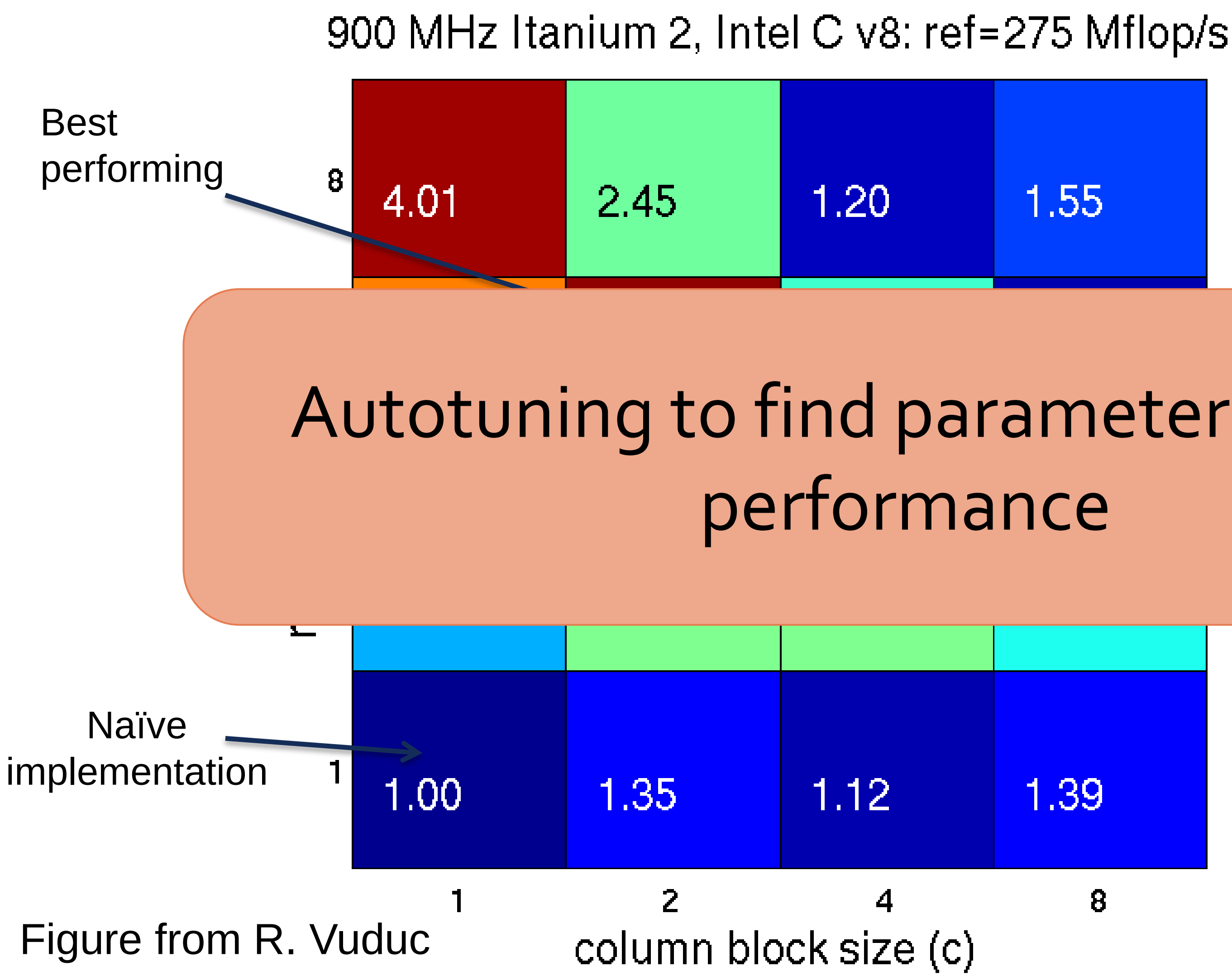
Intel Processor Clock Speed

[1] Krste Asanovic et al. "The Landscape of Parallel Computing Research: A View from Berkeley" December 2006

Writing Fast Code is Hard



Finding Best Implementation is Hard

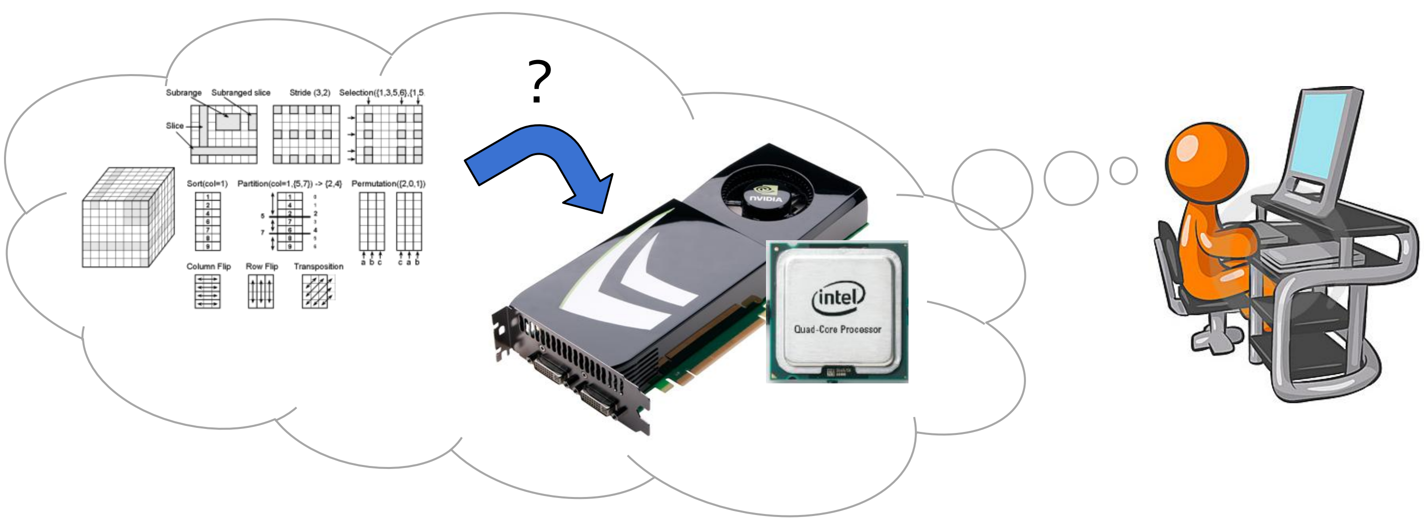


Autotuning to find parameters for best performance

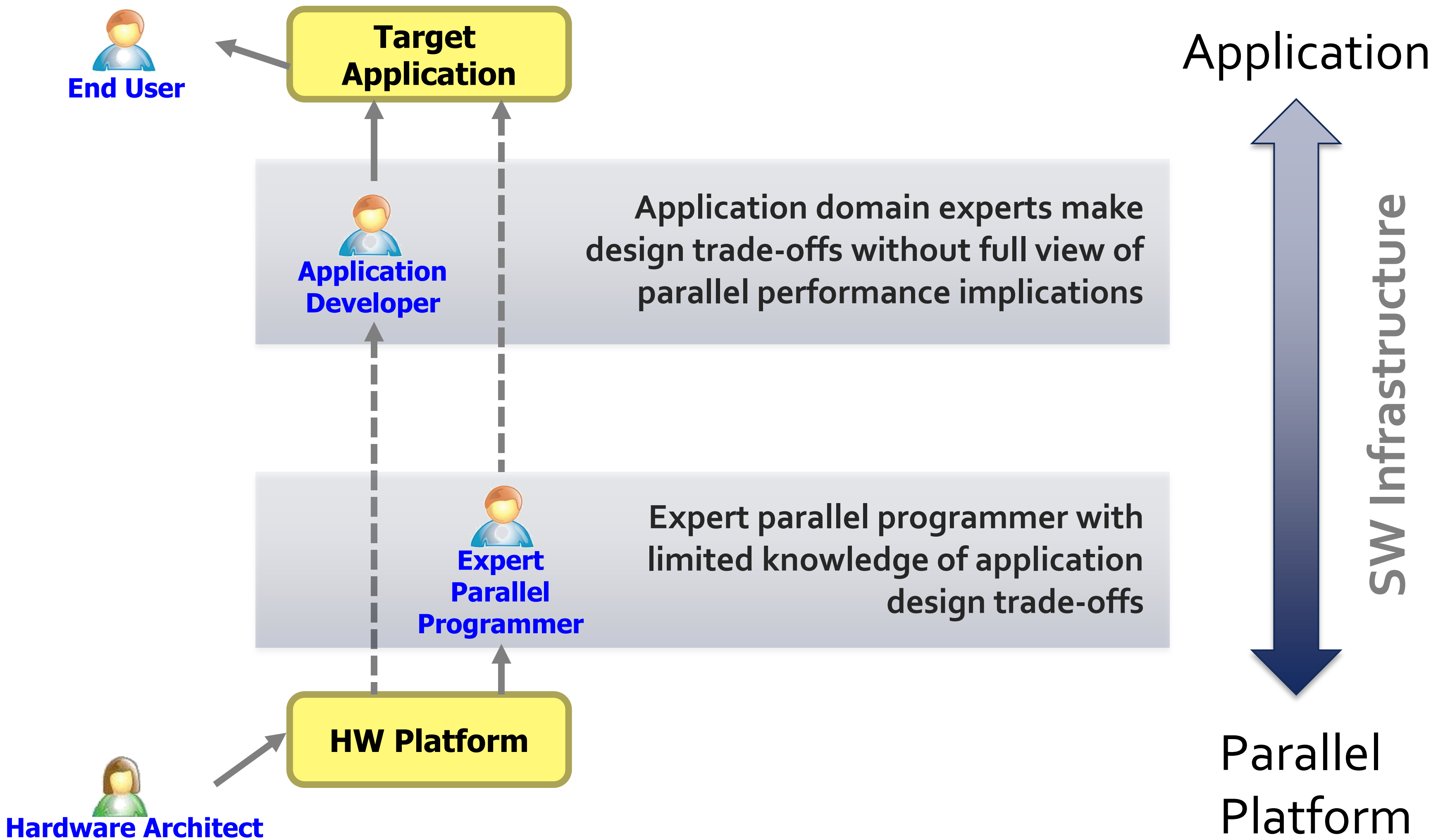
Figure from R. Vuduc

Productivity vs Performance

- Domain experts prefer to use high-level languages such as Python or MATLAB
- However, to achieve sufficient performance, computationally-intensive parts of applications must be rewritten in low-level languages
- Parallel platform and input parameters determine the best-performing parallel implementation



The Productivity-Performance Gap



Outline

1. Parallelism & productivity-performance gap
2. Proposed solution: Just-in-time specialization
3. Example: Gaussian mixture model (GMM) training specializer
4. Example applications using GMM specializer:
   1. Speaker diarization
   2. Music recommendation system
5. Summary
6. Future Work

Selective Embedded Just-In-Time Specialization (SEJITS)

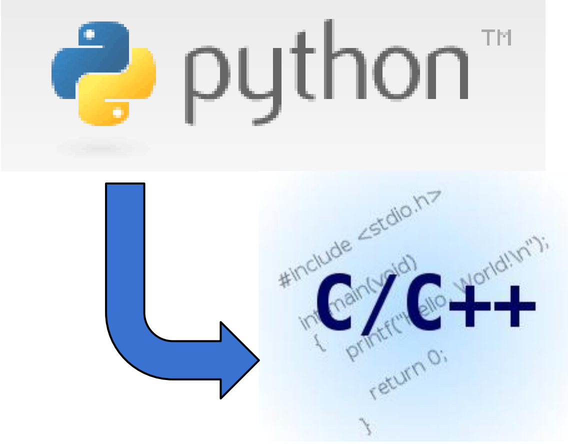
Key Idea: Generate, compile, and execute high performance parallel code at runtime using code transformation, introspection, variant selection and other features of high-level languages [2].

Invisibly to the user.

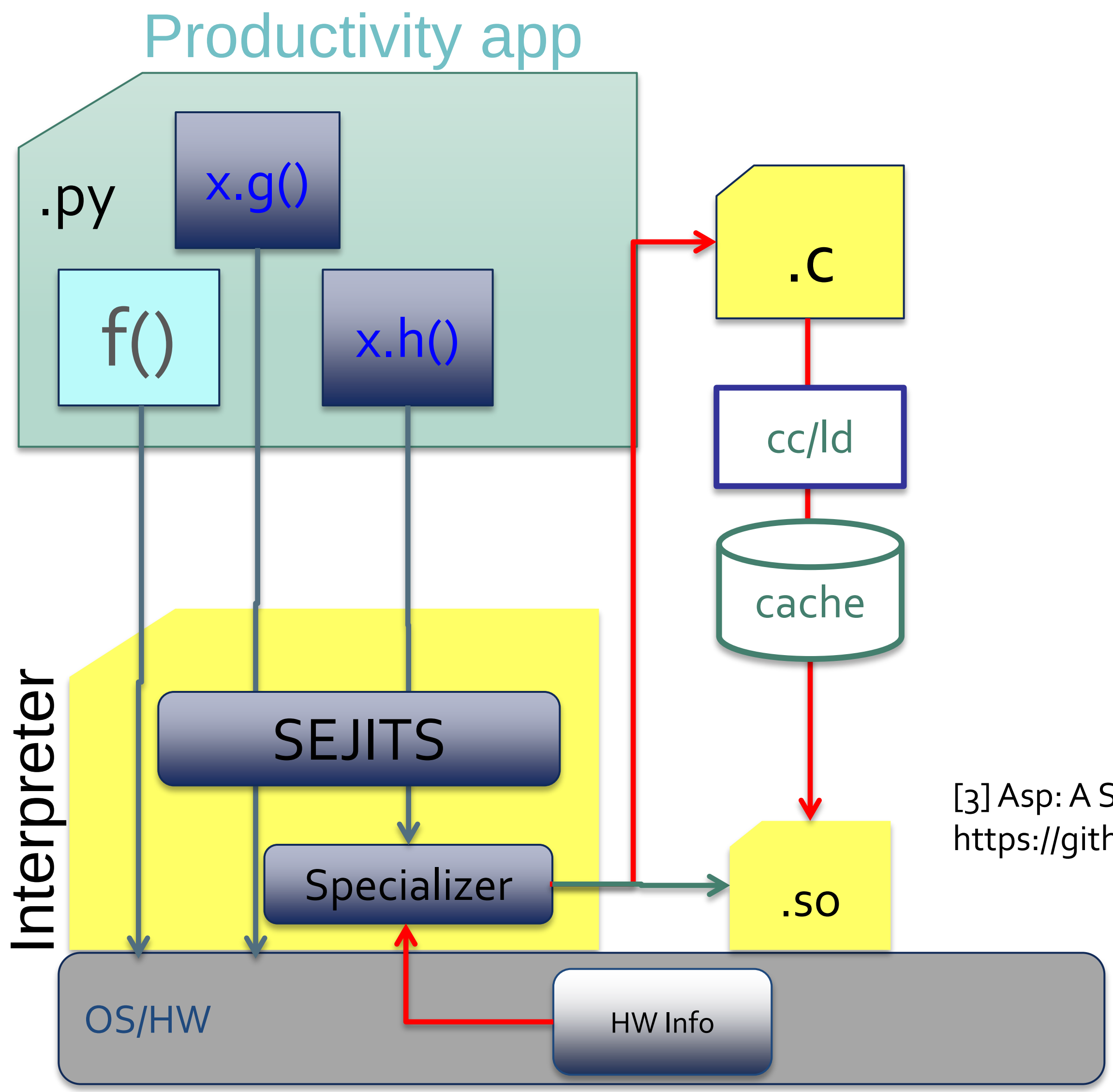
[2] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In Workshop on Programming Models for Emerging Architectures (PMEA 2009), Raleigh, NC, October 2009.

Selective Embedded JIT Specialization (SEJITS)

- Productivity-level language (PLL), e.g. Python for applications
- "Specializers" generate efficiency-level language (ELL) code targeted to hardware
  - Specialize specific computation
  - Code generation happens at runtime
  - Specializers can incorporate autotuning
- ELL performance with PLL effort



Selective Embedded JIT Specialization (SEJITS)



Asp – A SEJITS for Python [3]

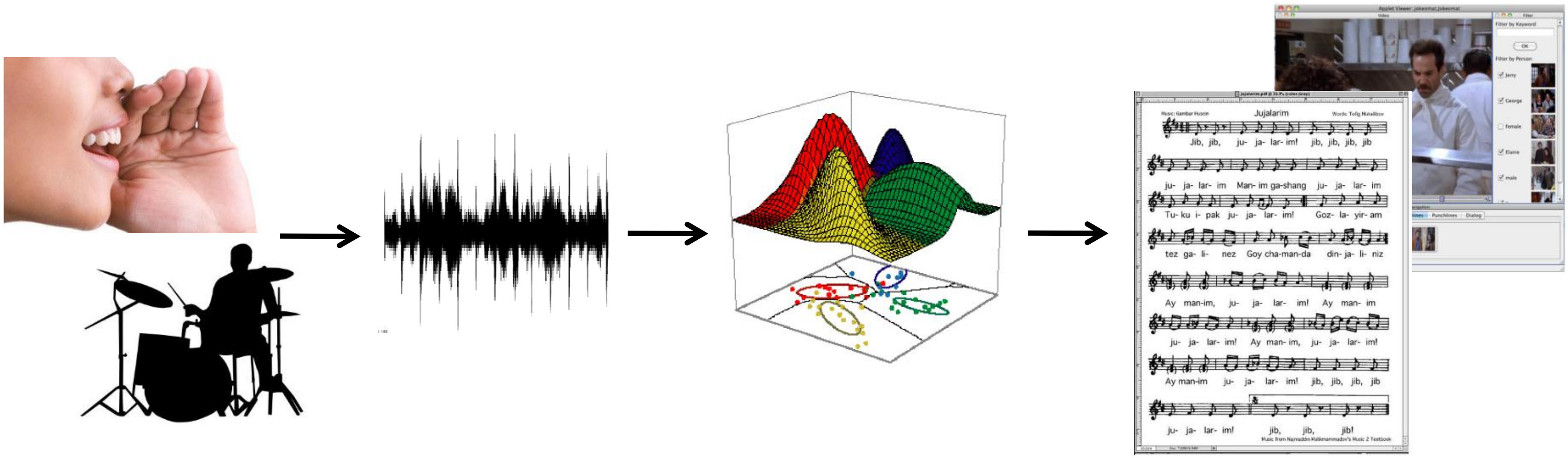
[3] Asp: A SEJITS implementation for Python. https://github.com/shoaibkamil/asp

Impact for programmers

- For productivity programmers
  - Efficient performance from high-level language
  - Further improvements in performance as specializers are added/refined
  - More programmers can exploit parallel architectures
  - Application code far more *portable & maintainable*
- For parallel programming experts
  - Provide useful common infrastructure for creating fast specializers
  - Wider impact & code reuse

Audio Content Analysis Applications

- Pattern recognition and information extraction from audio files



- Have impact on a big market
- Are computationally demanding
- Require processing large sets of data
- Have specific throughput and real-time constraints

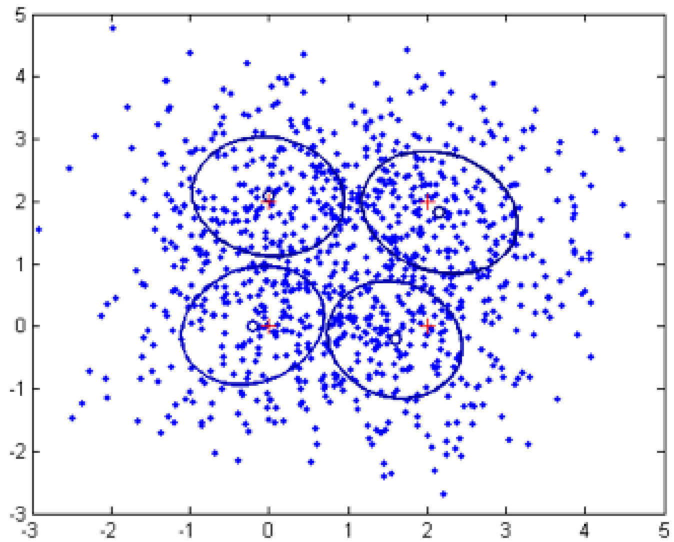
Outline

1. Parallelism & productivity-performance gap
2. Proposed solution: Just-in-time specialization
3. Example: Gaussian mixture model (GMM) training specializer
4. Example applications using GMM specializer:
    1. Speaker diarization
    2. Music recommendation system
5. Summary
6. Future Work

Gaussian Mixture Model (GMM)

- Probabilistic model for clustering 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: www.mathworks.com)



$$p(x_j \mid \mu_i, \Sigma_i) \;= \sum_i \pi_i \frac{1}{(2\pi)^{\frac{D}{2}} \mid \Sigma_i \mid^{\frac{1}{2}}} exp\{-\frac{1}{2}(x_j - \mu_i)^T \Sigma_i^{-1}(x_j - \mu_i)\}$$

GMM Training using EM Algorithm

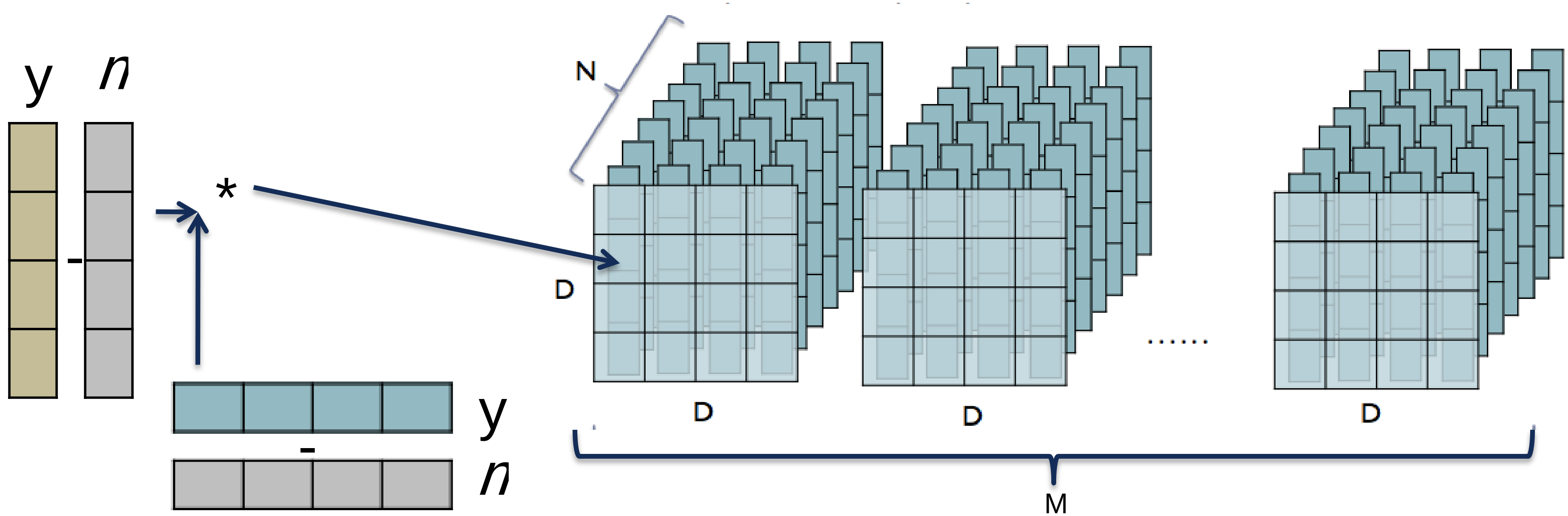
- Given a set of observations/events – find the maximum likelihood estimates of the set of Gaussian Mixture parameters (μ, Σ, p) and classify observations
- Expectation Maximization (EM) Algorithm
  - E step
    - Compute probabilities of events given model parameters
  - M step
    - Compute model parameters given probabilities
    - weights, mean, **covariance matrix**
  - Iterate until convergence
- Covariance matrix – most computationally intensive step

Based on original GPU implementation by
[4]. A. D. Pangborn. Scalable data clustering using gpus. Master's thesis, Rochester Institute of Technology, 2010.

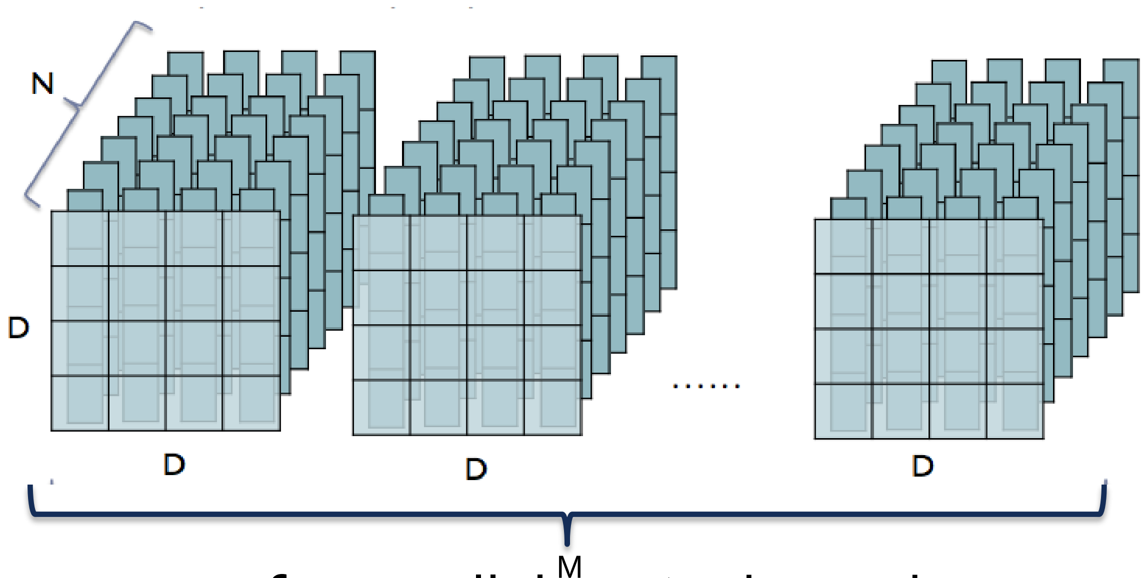
Covariance Matrix Computation

- N – number of feature vectors, ~10K-1M
- D – feature vector dimension, ~10-100
- M – number of Gaussian components, ~1-128
- Matrix is symmetric – only compute the lower D*D/2 cells

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



Covariance Matrix Computation



- Opportunities for parallelism (independent computations):
  - Each component's covariance matrix
  - Each cell in a covariance matrix
  - Each feature vector's contribution to a cell in a covariance matrix
- -> **Multiple code variants** to perform the same computation in different ways (here: on Nvidia GPUs)

Nvidia CUDA Programming Model

- CUDA is a recent programming model, designed for
  - Manycore (GPU) architectures
  - Wide vector (SIMD*) parallelism
  - Scalability
- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchronization & data sharing between small groups of threads
- CUDA programs are written in C + extensions

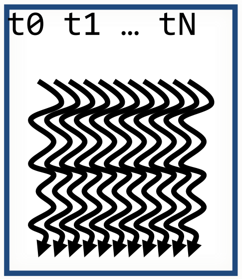
**SIMD = "Single Instruction, Multiple Data"*

Threads and Thread blocks

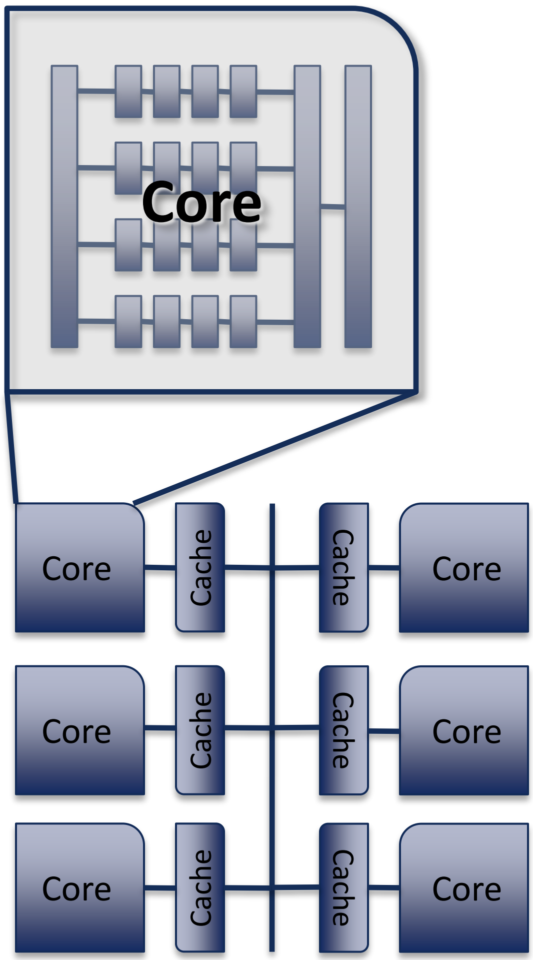
- Parallel kernels composed of many threads
  - all threads execute the same sequential program
  - Kernels:
    - Invoked from “Host” CPU code (C)
    - Executed on the “Device” GPU
- Threads are grouped into thread blocks
  - threads in the same block can cooperate
- Threads/blocks have unique IDs

Thread t

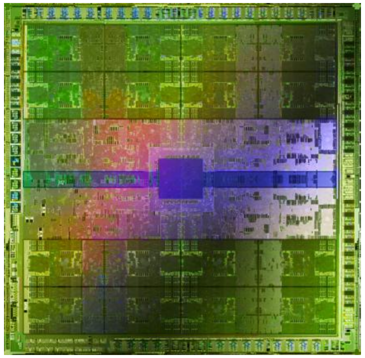
Block b



Manycore Parallel Platform



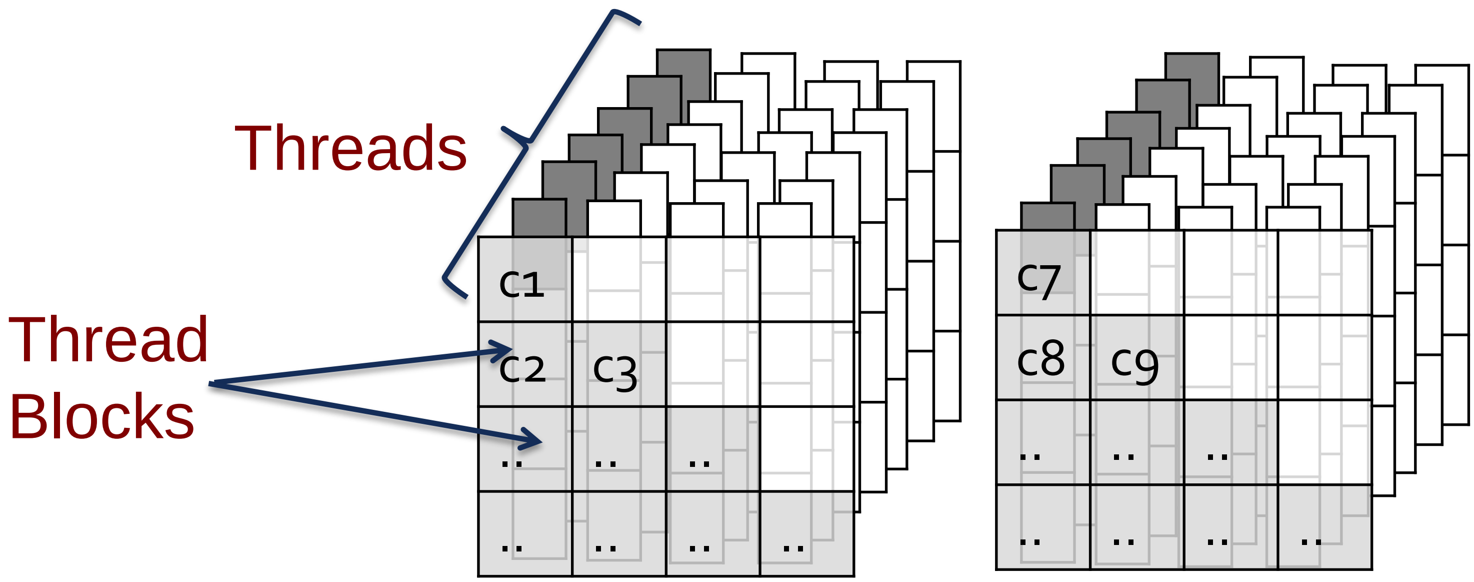
- Two levels of parallelism:
  - Cores
    - CUDA thread block
  - SIMD vector lanes within the core
    - CUDA threads
- Per-core local memory
  - Software Programmable
  - Shared by all threads in a thread block



Nvidia GTX480 (Fermi) Die Photo

Covariance Matrix Code Variants - Example

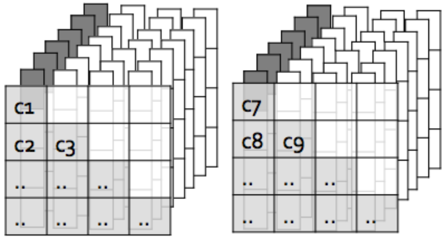
- Code variant 1:
  - 2D grid of thread blocks M x D*D/2
  - Each thread block is responsible for computing **one cell** in the covariance matrix for **one component**
  - Thread parallelization over feature vectors (N)



Covariance Matrix Computation – Code Variants

```
for each component m in M comps
  for each cell c in DxD/2 cells        → Thread block
    for each f n in N features          → Thread
      add nth contribution to c of m    → Seq.
```

V1




```
for each cell c in DxD/2 cells          → Thread block
  for each f n in N features            → Thread
    for each component m in M comps     → Seq.
      add nth contribution to c of m
```

V2

....

Covariance Matrix Computation – Code Variants Summary

V1

for each component m in M comps
  for each cell c in DxD/2 cells → Thread block
    for each f n in N features → Thread
      add nth contribution to c of m → Seq.

V2

for each cell c in DxD/2 cells → Thread block
  for each f n in N features → Thread
    for each component m in M comps
    add nth contribution to c of m → Seq.

V3

for each component m in M comps → Thread block
  for each cell c in DxD/2 cells → Thread
    for each f n in N features
    add nth contribution to c of m → Seq.

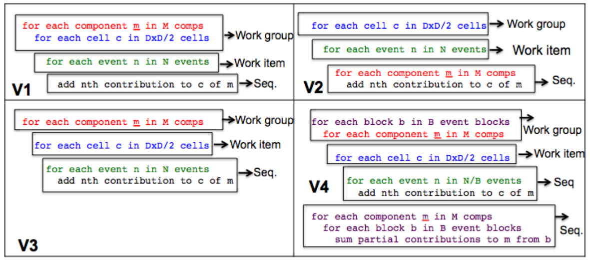
V4

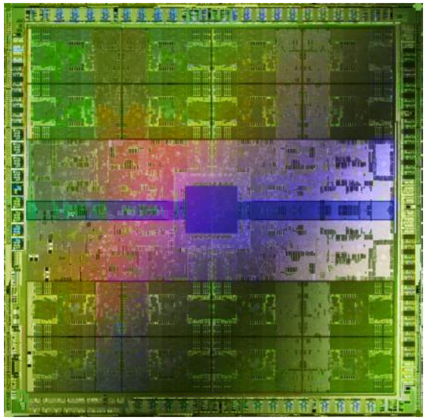
for each block b in B feature blocks
  for each component m in M comps → Thread block
    for each cell c in DxD/2 cells → Thread
      for each f n in N/B features
      add nth contribution to c of m → Seq

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

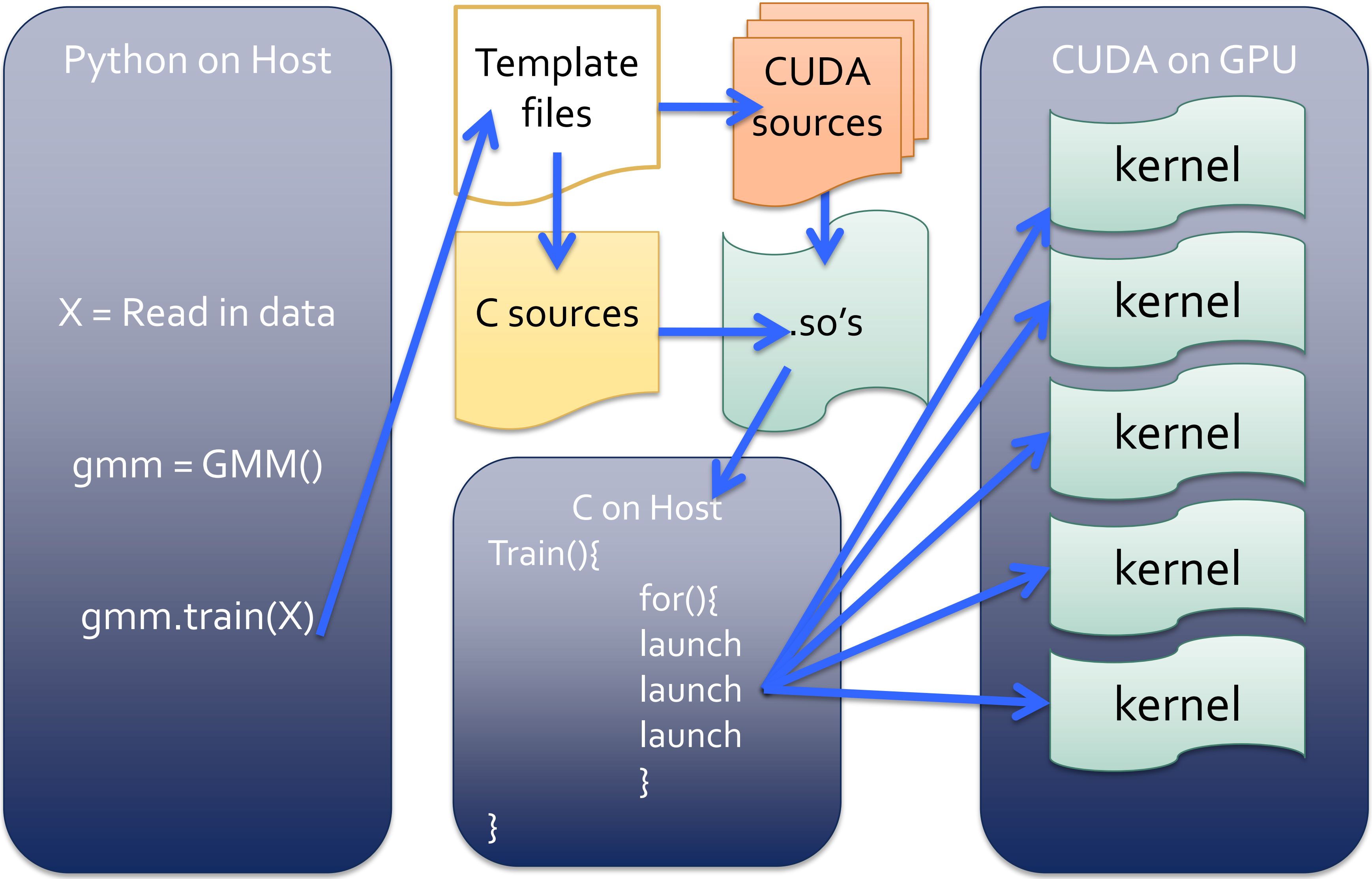
Specialization

- Given:
  - Problem Dimensions (N, D, M)
  - Platform Parameters (targeting Nvidia GPUs)
    - Core count, local memory size, SIMD width…
- Automatically select:
  - Optimal code variant
  - Optimal parameters (block size, number of blocks) for that code variant



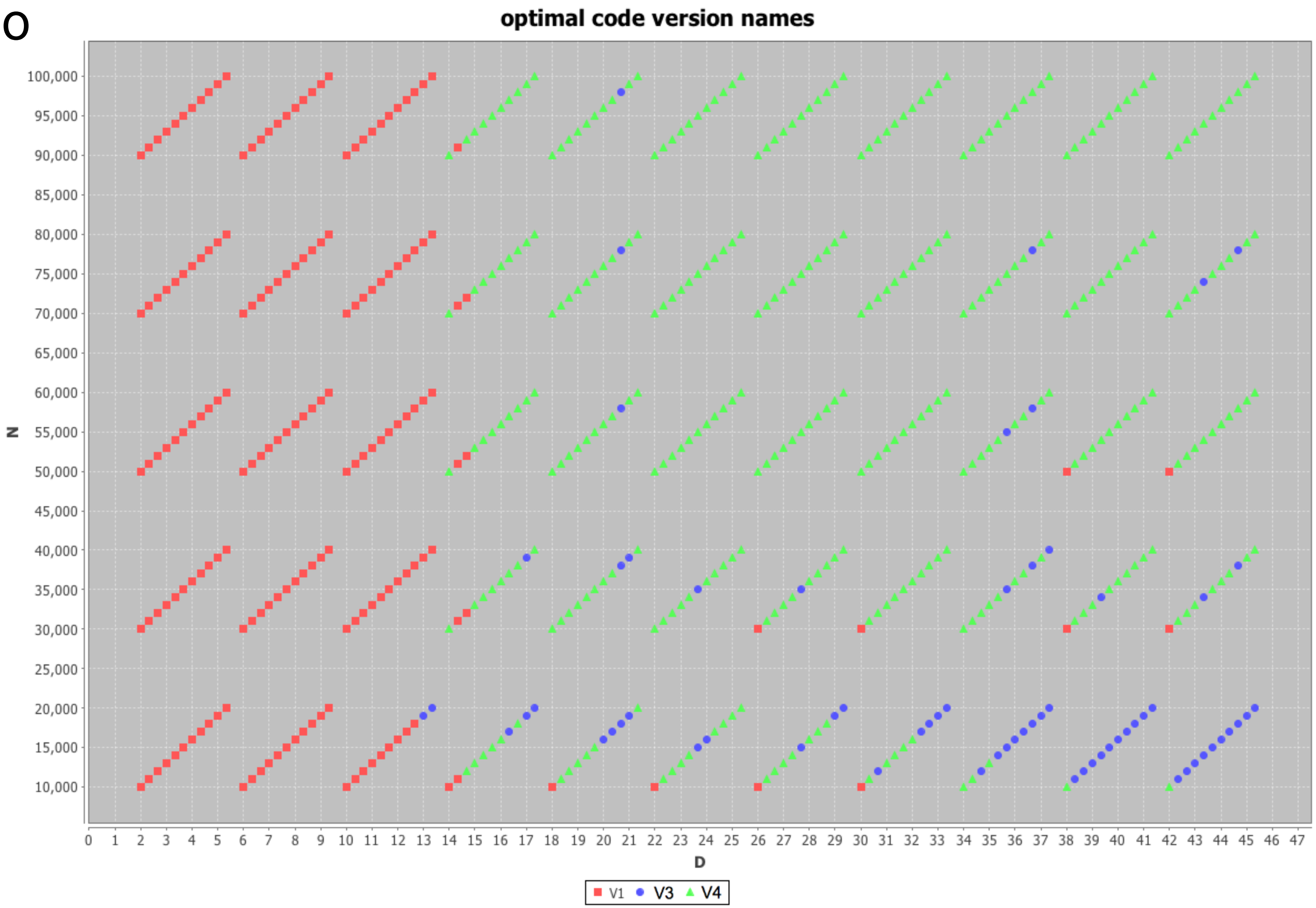


GMM Specializer: Overview



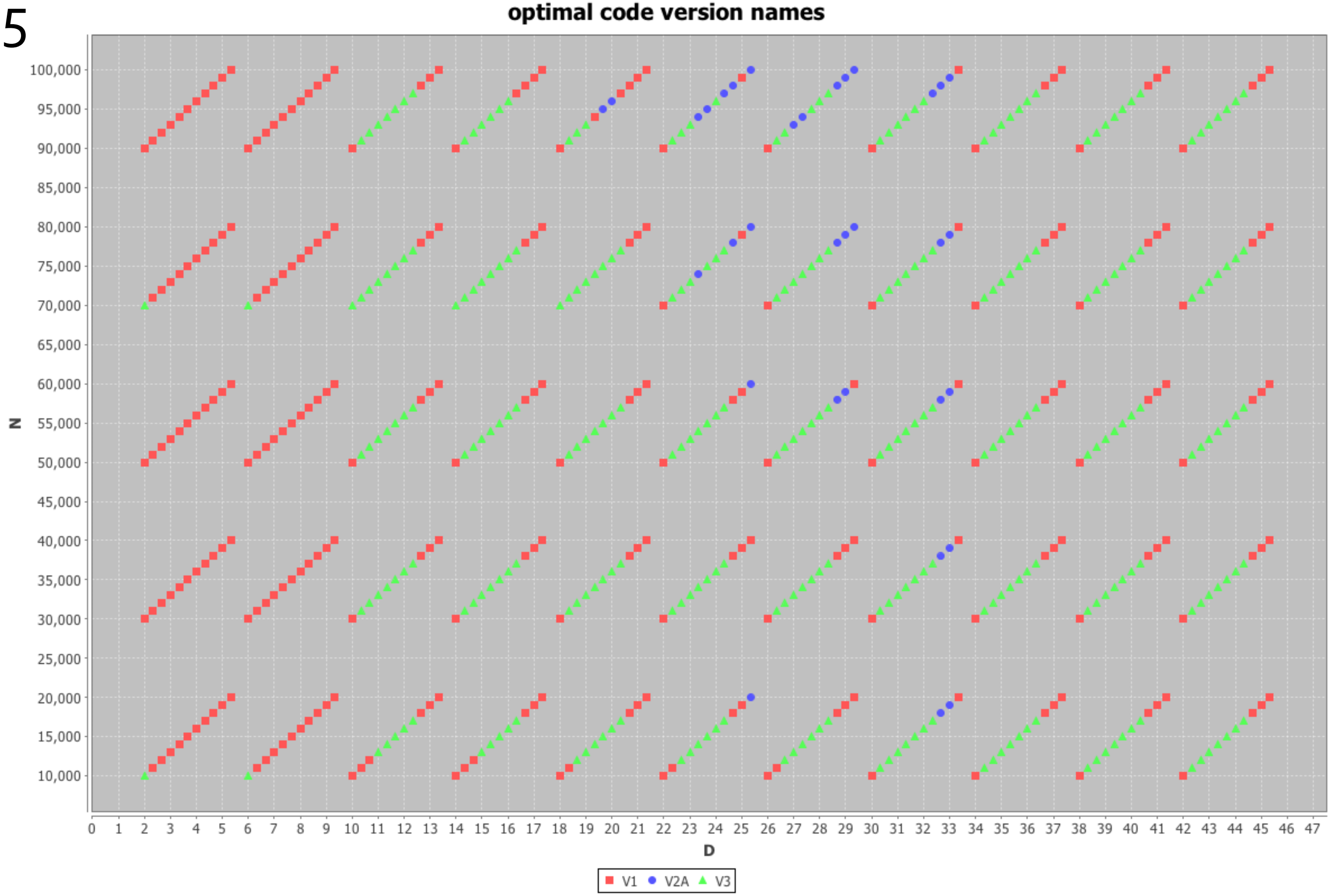
Results – Code Variant Performance

GTX480



Results – Code Variant Performance

GTX285



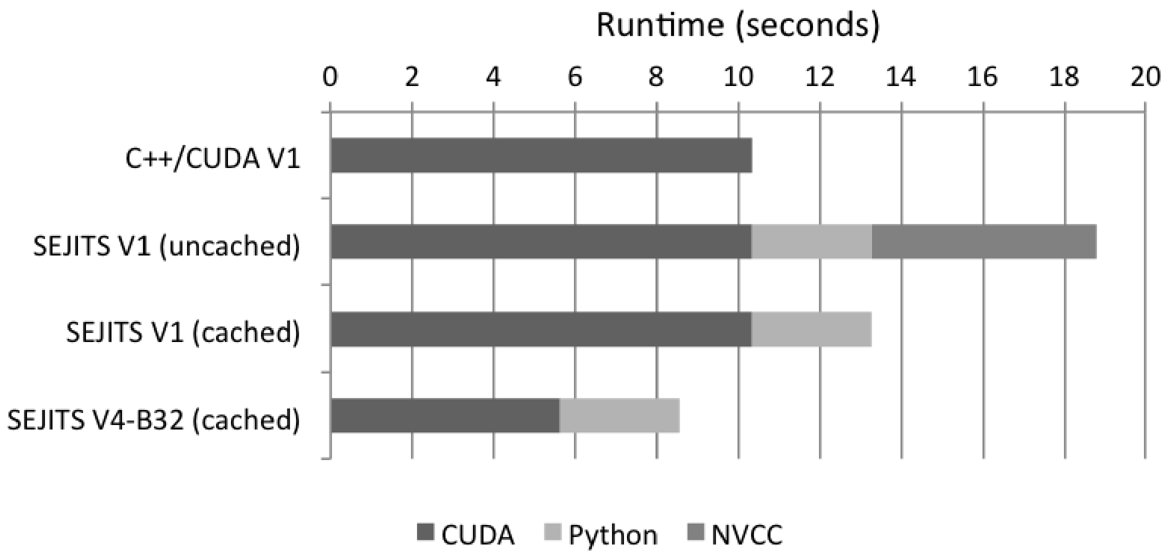
GPU	SMs	SIMD	Sh_mem Size	DRAM Size
GTX480	14	32	48KB	3GB
GTX285	30	8	16KB	1GB

Results - Code Variant Selection

- **32%** average improvement in covariance matrix computation time using best code variant
  - 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

Results – Specializer Overhead

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



Outline

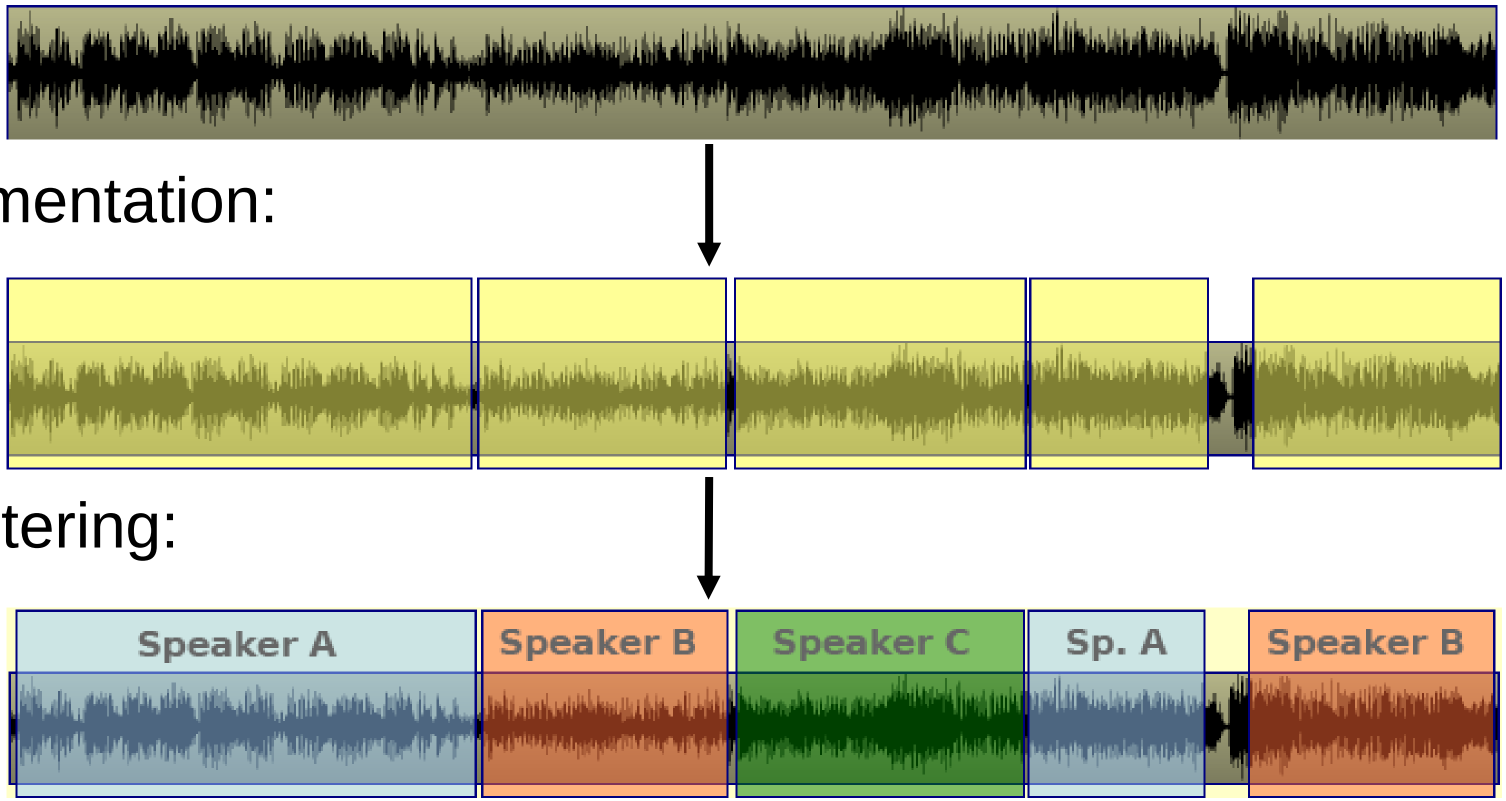
1. Parallelism & productivity-performance gap
2. Proposed solution: a Specialization Framework
3. Example: Gaussian mixture model (GMM) training specializer
4. Example applications using GMM specializer:
    1. Speaker diarization
    2. Music recommendation system
5. Summary
6. Future Work

Speaker Diarization

Audio track:

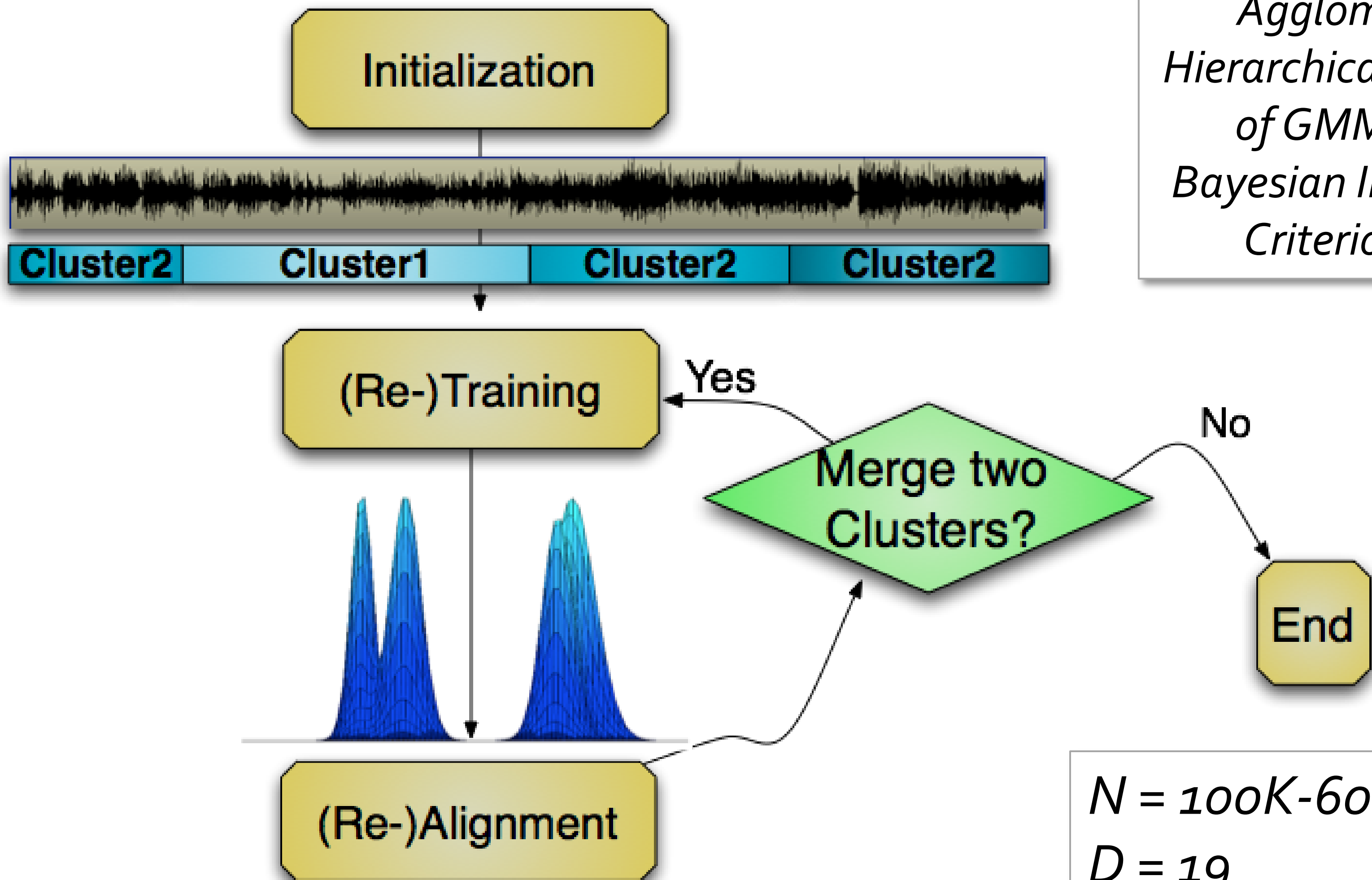
Segmentation:

Clustering:



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

Speaker Diarization: Core Algorithm

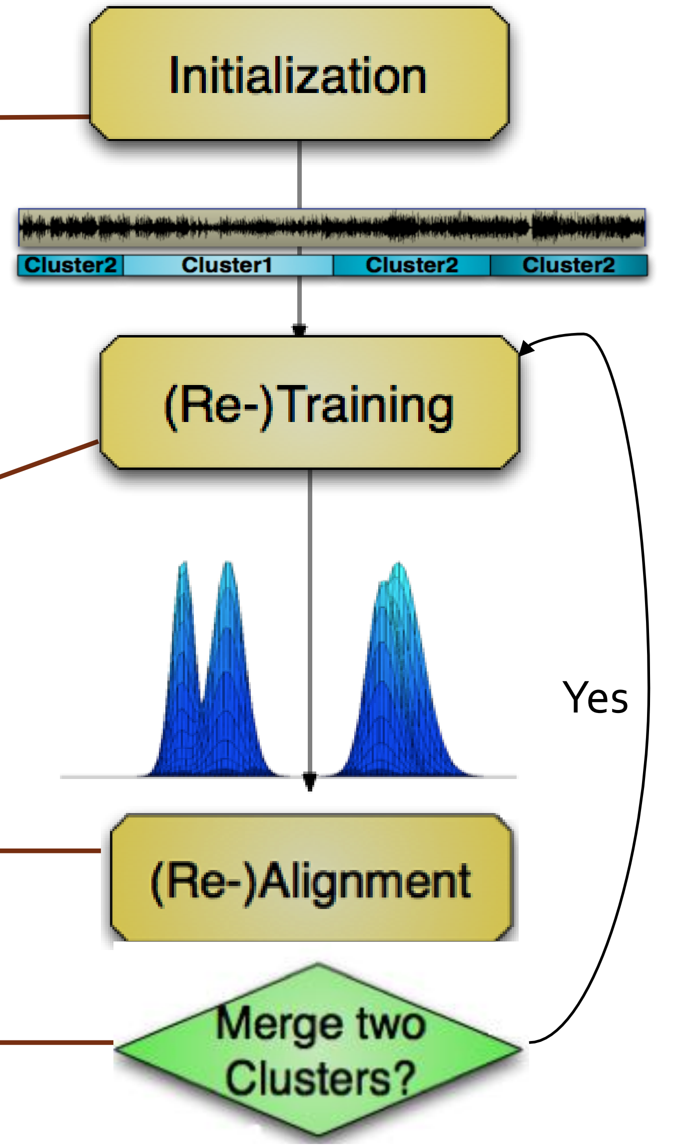


Agglomerative Hierarchical Clustering of GMMs using Bayesian Information Criterion (BIC)

N = 100K-600K
D = 19
M = 5-80

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):

    # Get the events, divide them into an initial k clusters and train each GMM on a cluster
    per_c
    init_                                                     cluster)))
    for g
        g
```

L = new_gmm_list(M,D)

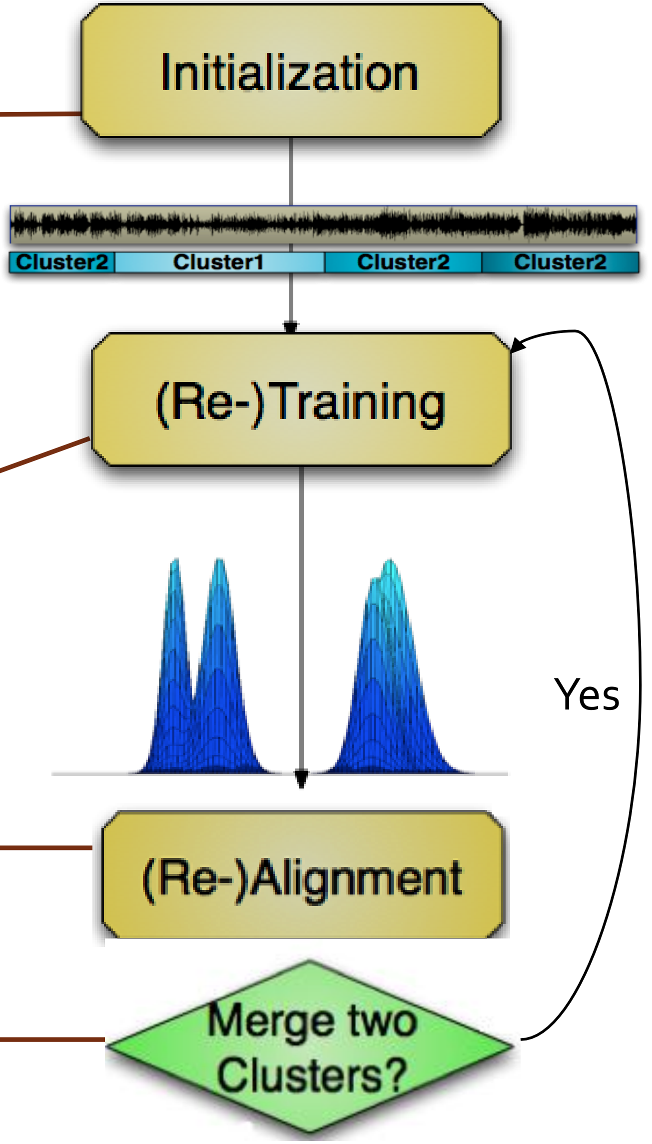
```
    # Perform hierarchical agglomeration based on BIC scores
    best_BIC_score = 1.0
    while (best_BIC_score > 0 and len(self.gmm_list) > 1):

        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)))

        # Across 2.5 secs of observations, vote on which cluster they should be associated with
        s
        f
```

for g in L : 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(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_                              atenate((d1, d2)))
                if s
                    best_BIC_score = score
```

g.train(x)

```
        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

Python

```python
def AHC(self):

    # Get the events, divide them into an initial k clusters and train each GMM on a cluster
    per_c
    init_                                                              N, per_cluster)))
    for g
        g

    # 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                                              t_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.scor
        most_likely = likelihoods.argmax(axis=1)

        # Across 2.5 secs of observations, vote on which cluster t
        split_events = split_events_based_on_votes(most_likely, se

        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((
                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 Lines-of-code reduction

Speaker Diarization Results

Diarization Error Rate (DER) and faster-than-real-time factor for the AMI Meeting Corpus

Average 71-115x Faster Than Real-Time Performance on NVIDIA Fermi GPU

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

[6] Ekaterina Gonina, Gerald Friedland, Henry Cook, Kurt Keutzer “Fast Speaker Diarization Using a High-Level Scripting Language” In Proceedings of IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), Dec 11-15, 2011, Waikoloa, Hawaii.

Results - Portability

Mic Array	Py+Cilk+	Py+CUDA
	Westmere	GTX285/GTX480
Near field	56×	101 × / 115×
Far field	32×	68 × / 71×

- Faster-than-real-time factors for:
  - Specializer on Intel Westmere (12 cores/24 threads)
  - Nvidia GTX280 & GTX480

GPU	SMs	SIMD	Sh_mem Size	DRAM Size
GTX480	14	32	48KB	3GB
GTX285	30	8	16KB	1GB

Outline

1. Parallelism & productivity-performance gap ✓
2. Proposed solution: a Specialization Framework ✓
3. Example: Gaussian mixture model (GMM) training specializer ✓
4. Example applications using GMM specializer:
    1. Speaker diarization ✓
    2. Music recommendation system
5. Summary
6. Future work

Content-based Music Recommendation: Pardora

- Given a query song or subset of songs – return similar songs
- Song recommendation system based on the *content* of the audio files
  - Audio segment-based features
- No need for tedious manual tagging!
- Can use any audio for querying
  - Your iTunes library?
  - Recording from a concert?
  - Humming your favorite song?

Dataset

- Million Song Dataset (MSD) from Columbia University: http://labrosa.ee.columbia.edu/millionsong/
- "A freely-available collection of audio features and metadata for a million contemporary popular music tracks"
- 1M song features & metadata
  - Artist & song information
  - Tags & beat information
  - MFCC-like timbre features

Pardora Recommendation System

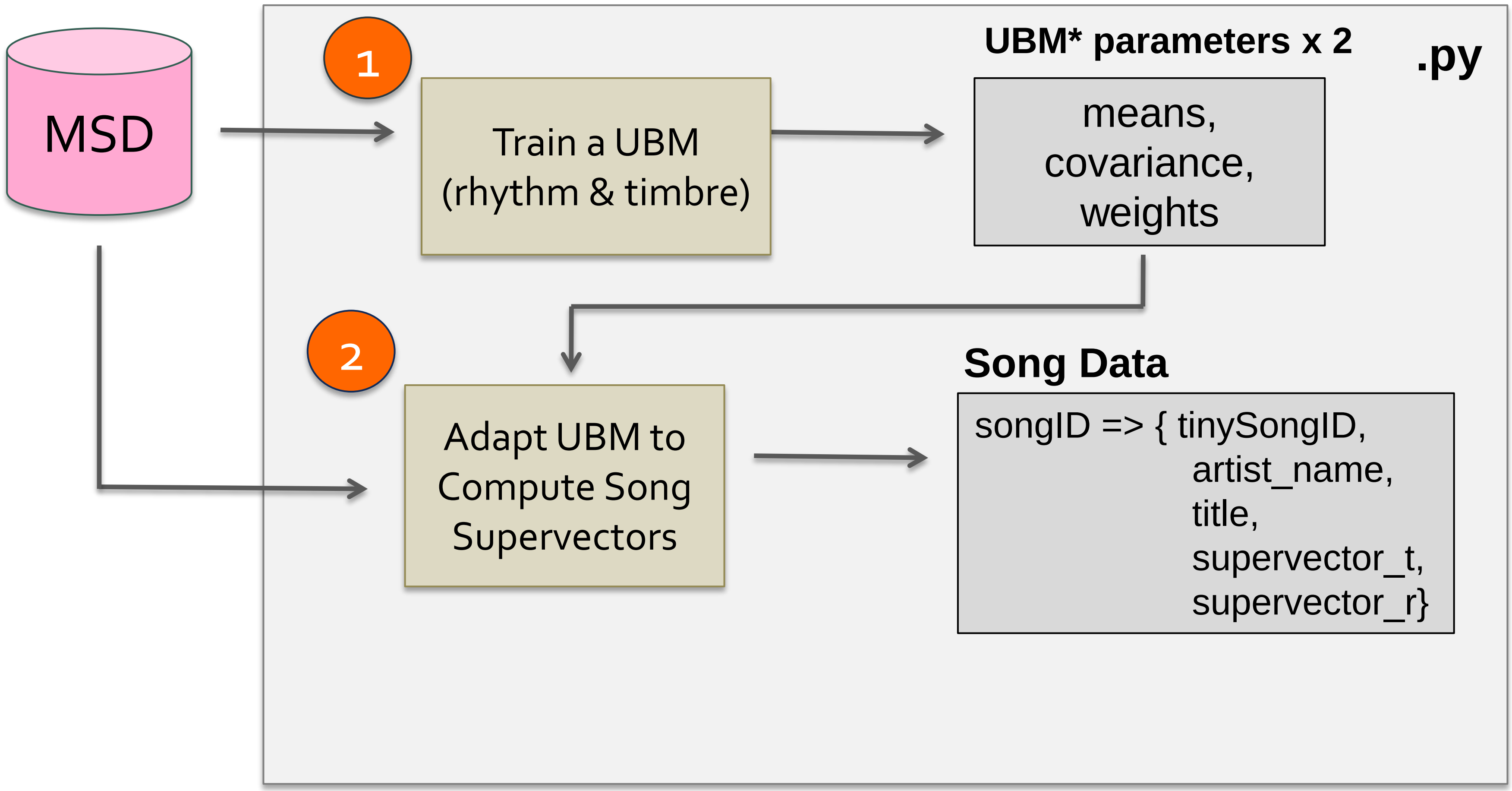
- Based on the UBM*-GMM supervector approach (IRCAM’10 [6]) (next slide)

1. Offline Phase: train UBM & song models
2. Online Phase: train query model & return top 10 closest songs

[6] C. Charbuillet, D. Tardieu, F. Cornu, and G. Peeters, “2011 IRCAM AUDIO MUSIC SIMILARITY SYSTEM#,” 2011.

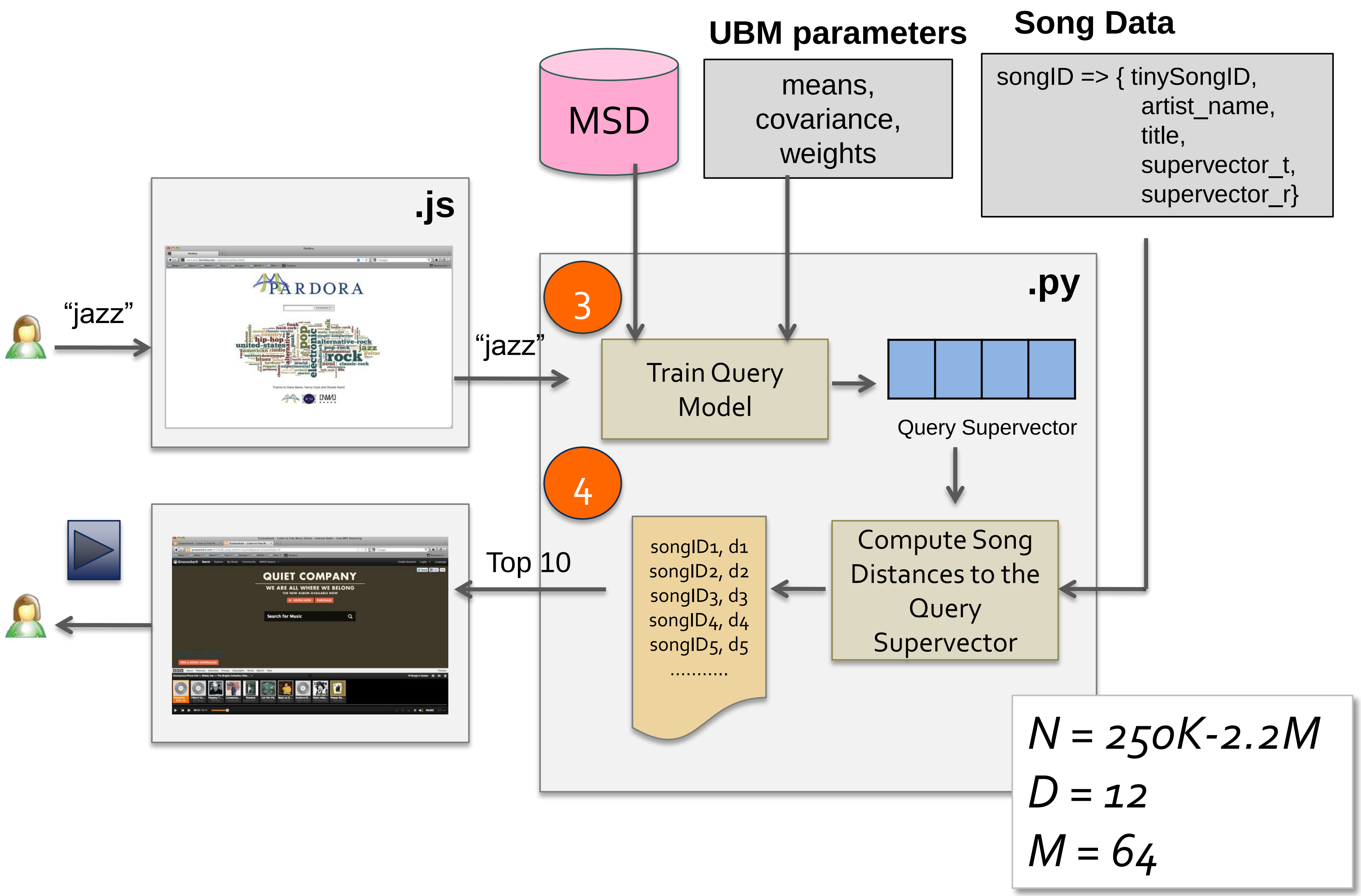
UBM* = Universal Background Model

Pardora – Offline Phase



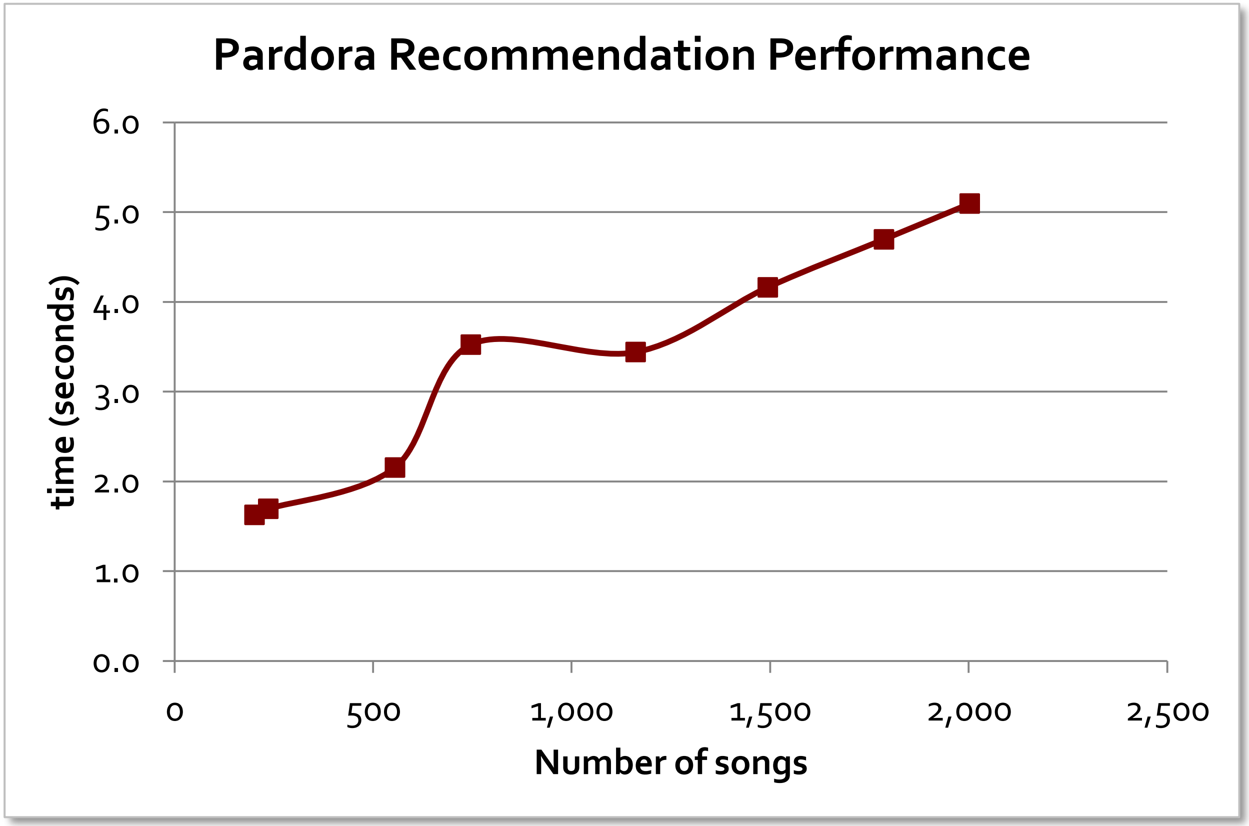
UBM* = Universal Background Model

Pardora – Online Phase



Pardora Performance Results

- Offline Phase: ~10 minutes
- Online Phase: 1.5-5 seconds depending on query size



Outline

1. Parallelism & productivity-performance gap
2. Proposed solution: a Specialization Framework
3. Example: Gaussian mixture model (GMM) training specializer
4. Example applications using GMM specializer:
   1. Speaker diarization
   2. Music recommendation system
5. Summary
6. Future work

Summary

- Programming parallel processors is challenging
- Selective JIT specialization can allow us to bridge the productivity-performance gap
- Example: Gaussian Mixture Model specializer
  - Python-level productivity & CUDA-level performance
- Two example applications:
  - Speaker Diarization (~100 lines of Python)
    - 71-115x faster-than-real-time performance
  - Music Recommendation System (~600 lines of Python)
    - Order of seconds for online recommendation
  - Productivity meets performance

Future Work

- Scalability of this approach for application development
  - More applications using more specializers
  - Focus on productivity & performance
- Scalability to the cloud for large datasets
  - Whole 1M songs will require cluster-level parallelism
- Autotuning and smarter code generation/selection
  - Incorporate machine learning & heuristics
- Specializer composition
  - Optimization & data structure selection

Thank you!

Research supported by 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.

Backup Slides

SEJITS Example: Structured Grids

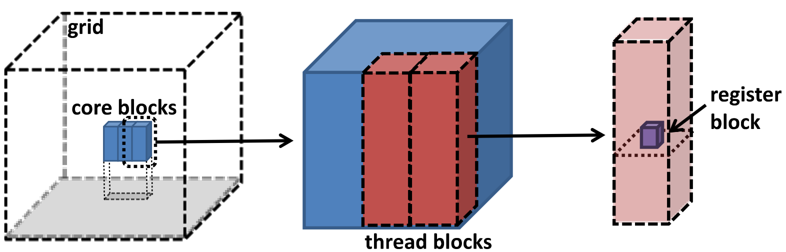
- User writes code for a structured grid calculation

```
# 3d heat equation
def kernel(inArray, outArray):
  for pt in inArray.interior():
    for x in pt.neighbors(radius=1):
      outArray[pt] += 1/6 * inArray[x]
```

SEJITS Example: Structured Grids

- When the user runs kernel(A,B):
  - Python code is transformed into optimized C code (more on that later)
    - Take into account # of cores, size of array (256^3)

```
int c2;
for (c2=chunkOffset_2;c2<=255;c2+=128) {
 int c1;
 for (c1=chunkOffset_1;c1<=255;c1+=64) {
  int c0;
  for (c0=chunkOffset_0;c0<=255;c0+=256) {
   int b2;
   for (b2=c2 + threadOffset_2;b2<=c2 + 127;b2+=128) {
    int b1;
    for (b1=c1 + threadOffset_1;b1<=c1 + 31;b1+=16) {
     int b0;
     for (b0=c0 + threadOffset_0;b0<=c0 + 255;b0+=256) {
      int kk;
      for (kk=b2 + 1;kk<=b2 + 128;kk+=1) {
       int jj;
       for (jj=b1 + 1;jj<=b1 + 16;jj+=1) {
        int ii;
        for (ii=b0 + 1;ii<=b0 + 256;ii+=1) {
         dst[_dst_Index(ii - 1,jj - 1,kk - 1)] = ...;
}}}}}}}}}
```

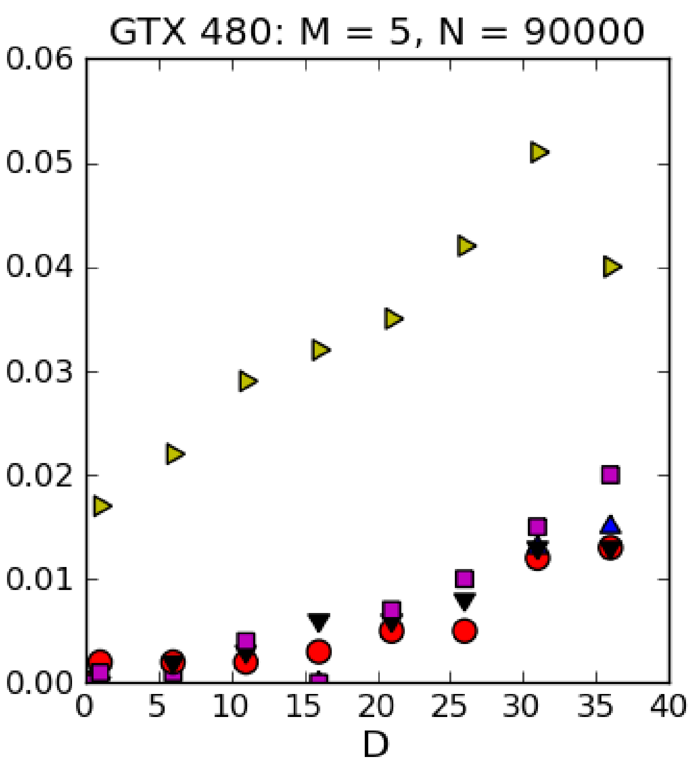


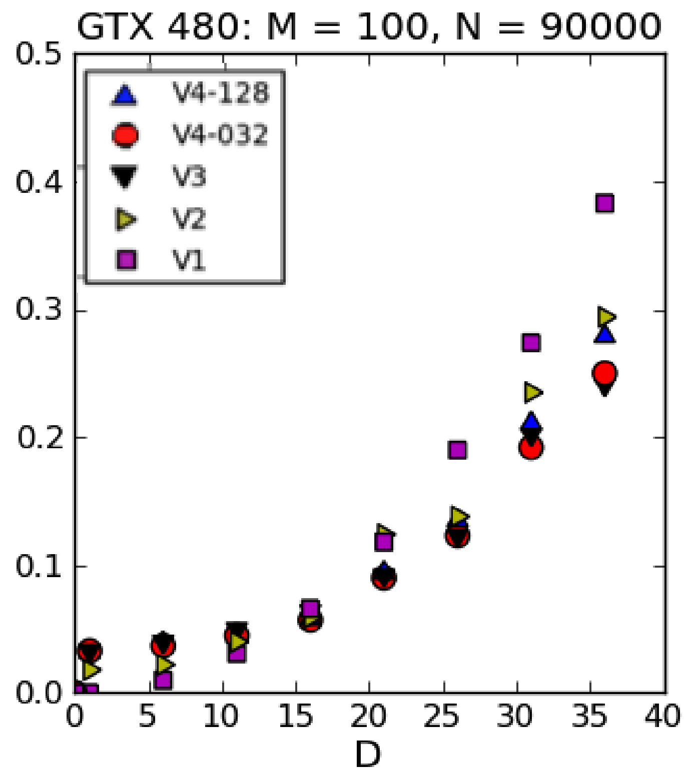
SEJITS Example: Structured Grids

- When the user runs kernel(A,B):
  - Python code is transformed into optimized C code (more on that later)
  - Code is output to disk
  - Compiler runs, turns it into dynamic library
  - Library is loaded into the interpreter
  - Translated function is called & result returned to interpreter
- To user, it just looks like the code ran really fast

Results – Version Comparison (Raw CUDA)

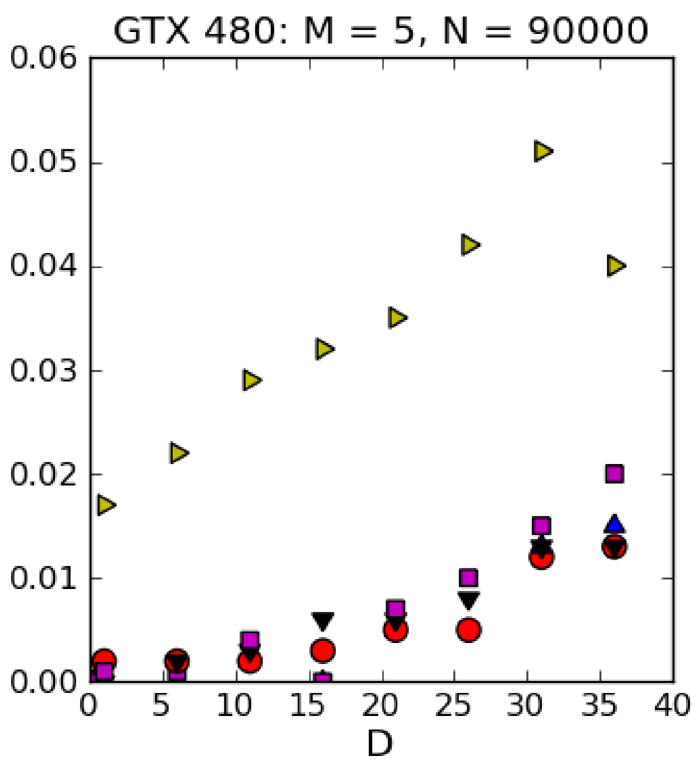
NVIDIA GTX480 – Varying D

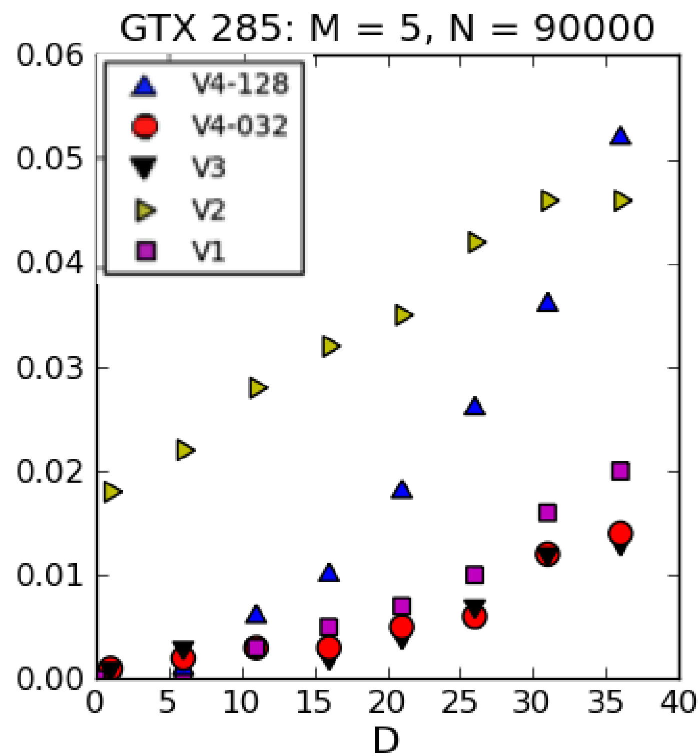




Results – Version Comparison (Raw CUDA)

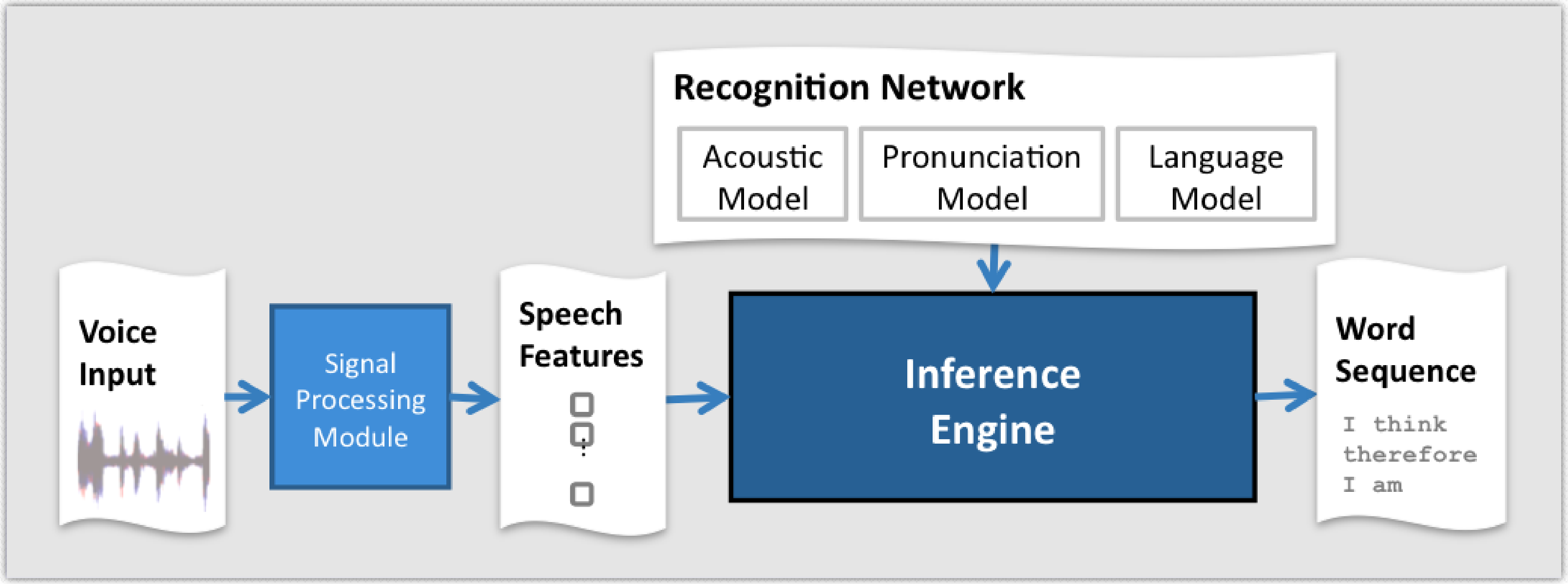
NVIDIA GTX285 vs. 480





GPU	SMs	SIMD	Sh_mem Size	DRAM Size
GTX480	14	32	48KB	3GB
GTX285	30	8	16KB	1GB

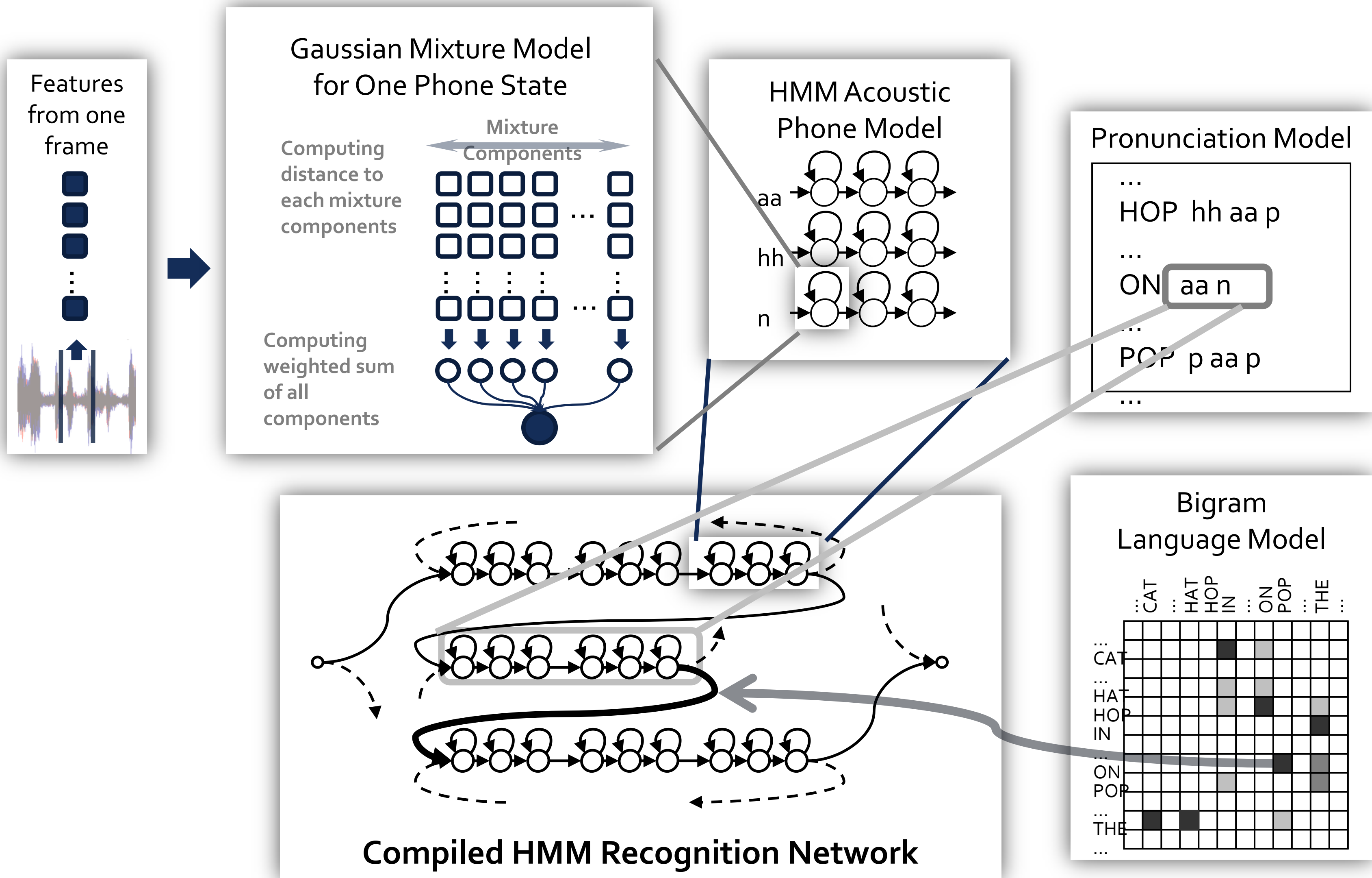
Example: Speech Recognition



- Task: recognize words and sentences from an audio file
  - Recognizing words from a large vocabulary arranged in exponentially many possible permutations
  - Inferring word boundaries from the context of neighboring words
- Viterbi decoding on Hidden Markov Models

Jike Chong, Ekaterina Gonina, Kurt Keutzer, "Efficient Automatic Speech Recognition on the GPU" Chapter in GPU Computing Gems Emerald Edition, Morgan Kaufmann, Vol. 1, February 9, 2011.

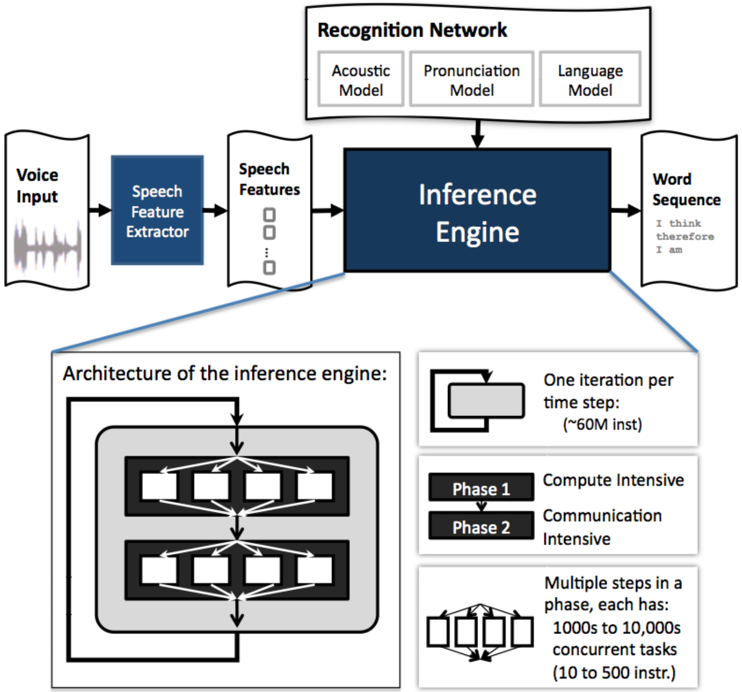
Example: Speech Recognition

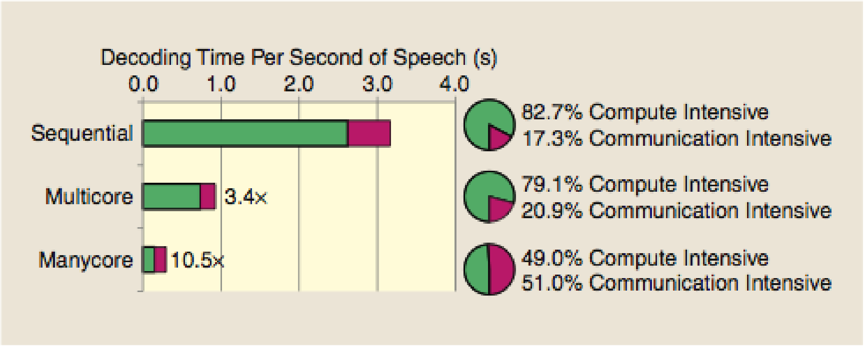


My Previous Work (1)

Fully-parallel Speech Recognition Decoder

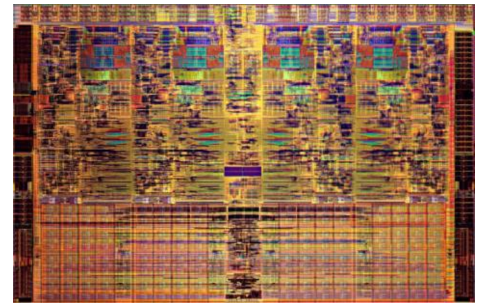
- Efficient multicore and manycore implementations of entire decoder (InterSpeech'09)
- Exploring
  - Algorithmic-level design space (IEEE SP Journal 2009)
  - Recognition network representation (InterSpeech'11)



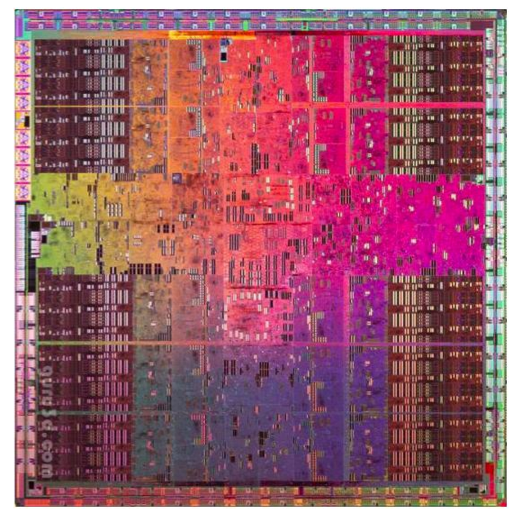


Multicore & Manycore, *cont.*

Specifications	Core i7 960	GTX285
Processing Elements	4 cores, 4 way SIMD @3.2 GHz	30 cores, 8 way SIMD @1.5 GHz
Resident Strands/Threads (max)	4 cores, 2 threads, 4 way SIMD: 32 strands	30 cores, 32 SIMD vectors, 32 way SIMD: 30720 threads
SP GFLOP/s	102	1080
Memory Bandwidth	25.6 GB/s	159 GB/s
Register File	-	1.875 MB
Local Store	-	480 kB



Core i7 (45nm)



GTX285 (55nm)

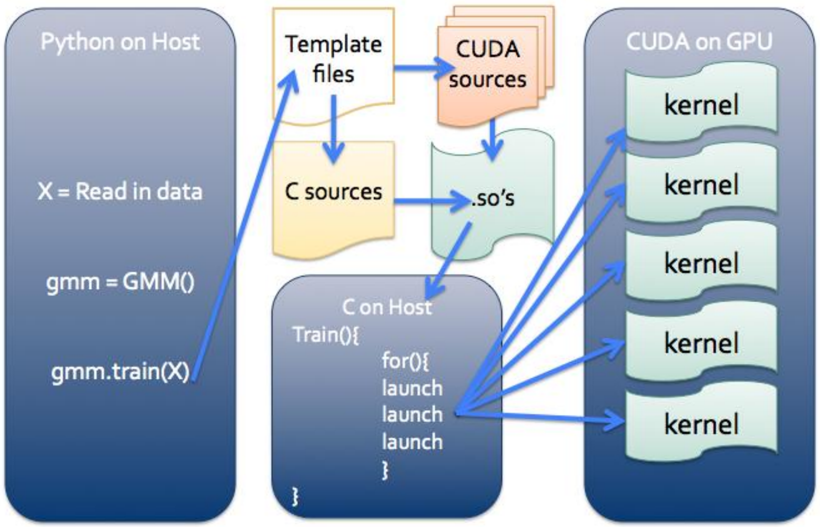
SIMD

- Single Instruction Multiple Data architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

GMM Specializer: Details

- Python application code
  - Manipulates problem data, sets up application logic
- C/CUDA code that runs quickly
  - Allocates GPU memory
  - Performs main EM iterative loop

- Specializer [5]
  - Selects appropriate code variant (from history) based on parameters
  - Pulls in the template for the code variant, parameterizes it and compiles to binary



[5] H. Cook, E. Gonina, S. Kamil, G. Friedland, D. Patterson, A. Fox. "CUDA-level Performance with Python-level Productivity for Gaussian Mixture Model Applications" In Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar'11). USENIX Association, Berkeley, CA, USA.