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

CS294, September 19, 2012
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]

Writing Fast Code is Hard

Dense Matrix Multiply (V. Volkov)

Fraction of Arithmetic Peak

Dimension of Matrices

0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%

0 128 256 384 512 640 768

ACML (vendor-provided binary)
an optimized code
(unrolling, explicit vectorization, few levels of blocking)

naïve blocking
Finding Best Implementation is Hard

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

Best performing

Naïve implementation

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

Application domain experts make design trade-offs without full view of parallel performance implications

Expert parallel programmer with limited knowledge of application design trade-offs

Application

SW Infrastructure

Parallel Platform

Target Application

HW Platform

Application Developer

Expert Parallel Programmer

End User

Hardware Architect
1. Parallelism & productivity-performance gap
2. Proposed solution: Just-in-time specialization
3. Example: Gaussian mixture model (GMM) training specializer
4. Example application using GMM specializer:
   1. Music recommendation system
5. Summary
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.

1. Productivity-level language (PLL), e.g. Python for applications

2. “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
Asp – A SEJITS for Python [3]

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
1. Parallelism & productivity-performance gap

2. Proposed solution: Just-in-time specialization

3. Example: Gaussian mixture model (GMM) training specializer

4. Example application using GMM specializer:
   1. Music recommendation system

5. Summary
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 ($\mu$) and a covariance ($\sigma$) parameters
  - Gaussians in the mixture are weighted with weight $\pi$

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

Example GMM in two dimensions (Source: www.mathworks.com)
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 event probabilities
      - weights, mean, covariance matrix
  - Iterate until convergence

- Covariance matrix – most computationally intensive step

Based on original GPU implementation by
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 \( D \times D/2 \) 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”
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**
- 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
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

Python on Host

X = Read in data

gmm = GMM()

gmm.train(X)

CUDA on GPU

Template files

C sources

CUDA sources

C sources

.so’s

C on Host

Train() {

for () {

launch

launch

launch

}

}

CUDA on GPU

kernel

kernel

kernel

kernel

kernel
Results – Code Variant Performance

GTX480

optimal code version names
Outline

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


UBM* = Universal Background Model
Pardora – Offline Phase

MSD

1. Train a UBM

2. Adapt UBM to Compute Song Supervectors

UBM* parameters
- means,
- covariance,
- weights

Song Data
- songID => { tinySongID, artist_name, title, supervector}

UBM* = Universal Background Model
Pardora – Online Phase

UBM parameters
- means,
- covariance,
- weights

MSD

Song Data
- songID => { tinySongID, artist_name, title, supervector}

Train Query Model

Query Supervector

Song Data

Compute Song Distances to the Query Supervector

Top 10

songID1, d1
songID2, d2
songID3, d3
songID4, d4
songID5, d5
...........

N = 250K-2.2M
D = 12
M = 64
Pardora - Results

Recommendation Time vs. Query Size – 10K Songs

- Number of features in query
- Total recommendation time
- Query GMM training time

1-2 songs
“Elton John”

10-17 songs
“Elton John or Eric Clapton or Lady Gaga or Britney Spears”
1. Parallelism & productivity-performance gap
2. Proposed solution: a Specialization Framework
3. Example: Gaussian mixture model (GMM) training specializer
4. Example application using GMM specializer:
   1. Music recommendation system
5. Summary
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
- An example application:
  - Music Recommendation System (~600 lines of Python)
    - Order of seconds for online recommendation
    - Productivity meets performance
Thank you!

Code available at: https://github.com/hcook/gmm
Wiki: https://github.com/hcook/gmm/wiki/Using-the-GMM-Specializer

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
- Code variant 1:
  - 2D grid of thread blocks $M \times D^2/2$
  - Each thread block is responsible for computing one cell in the covariance matrix for one component
  - Thread parallelization over feature vectors ($N$)
for each component \( m \) in \( M \) comps
for each cell \( c \) in \( D \times D/2 \) cells

for each \( f \) in \( N \) features

add \( n \)th contribution to \( c \) of \( m \)

---

V1

---

V2
Covariance Matrix Computation – Code Variants Summary

- **V1**: For each component \( m \) in \( M \) comps
  - For each cell \( c \) in \( D \times D/2 \) cells
    - For each feature \( n \) in \( N \) features
      - Add \( n \)th contribution to \( c \) of \( m \)

- **V2**: For each component \( m \) in \( M \) comps
  - For each cell \( c \) in \( D \times D/2 \) cells
    - For each feature \( n \) in \( N \) features
      - Add \( n \)th contribution to \( c \) of \( m \)

- **V3**: For each component \( m \) in \( M \) comps
  - For each cell \( c \) in \( D \times D/2 \) cells
    - For each feature \( n \) in \( N \) features
      - Add \( n \)th contribution to \( c \) of \( m \)

- **V4**: For each component \( m \) in \( M \) comps
  - For each cell \( c \) in \( D \times D/2 \) cells
    - For each feature \( n \) in \( N/B \) features
      - Add \( n \)th contribution to \( c \) of \( m \)
  - For each block \( b \) in \( B \) feature blocks
    - For each component \( m \) in \( M \) comps
      - Sum partial contributions to \( m \) from \( b \)
Estimate “who spoke when” with no prior knowledge of speakers, #of speakers, words, or language spoken.
Speaker Diarization: Core Algorithm

- Start with too many clusters (initialized randomly)
- Purify clusters by comparing and merging similar clusters
- Resegment and repeat until no more merging needed

\[ N = 100K - 600K \]
\[ D = 19 \]
\[ M = 5 - 80 \]

Agglomerative Hierarchical Clustering of GMMs using Bayesian Information Criterion (BIC)
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
    while (len(gmm_list) > 1):
        num_clusters = len(gmm_list)
        # Resegment data based on likelihood scoring
        likelihoods = gmm_list[0].score(self.X)
        for g in 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
        merged_tuple = None
        for g1 in range(len(self.gmm_list)):
            for g2 in range(g1+1, len(self.gmm_list)):
                g1, d1 = self.gmm_list[g1]
                g2, d2 = self.gmm_list[g2]
                score = 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)
```
```python
def AHC(self):
    # Get the events, divide them into an initial k clusters and train each GMM on a cluster
    # Initialize the clusters
    L = new_gmm_list(M, D)
    
    # Perform hierarchical agglomeration based on BIC scores
    best_BIC_score = 0.0
    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)])
        mostlikely = np.argmax(likelihoods)

        # Across 2.5 secs of observations, vote on which cluster they should be associated with
        for g in L:
            g.train(x)  # g.train(x)
    
    # Score all pairs of GMMs using BIC
    best_merged_gmm = None
    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
            if s:
                g.train(x)
            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)
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_cluster = self.N/self.init_num_clusters
    init_training = zip(self.gmm_list, np.vsplit(self.X, range(self.N, self.N, per_cluster)))
    for g, x in init_training:
        g.train(x)
    # Perform hierarchical agglomeration based on BIC scores
    best_BIC_score = 0.0
    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
        split_events = split_events_based_on_votes(most_likely, self)
        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 gmlidx in range(len(self.gmm_list)):
            for gmlidx1 in range(gmlidx+1, len(self.gmm_list)):
                g1, d1 = self.gmm_list[gmlidx]
                g2, d2 = self.gmm_list[gmlidx1]
                score = 0.0
                new_gmm, score = compute_distance_BIC(g1, g2, np.concatenate([d1, d2]))
                if score > best_BIC_score:
                    best_BIC_score = score
                    best_merged_gmm = new_gmm
                    merged_tuple = (g1, g2)
        # 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)
```

15x Lines-of-code reduction
## Speaker Diarization Results

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

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

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

Results - Portability

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

<table>
<thead>
<tr>
<th>Mic Array</th>
<th>Py+Cilk+</th>
<th>Py+CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Westmere</td>
<td>GTX285/GTX480</td>
</tr>
<tr>
<td>Near field</td>
<td>56×</td>
<td>101× / 115×</td>
</tr>
<tr>
<td>Far field</td>
<td>32×</td>
<td>68× / 71×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU</th>
<th>SMs</th>
<th>SIMD</th>
<th>Sh_mem Size</th>
<th>DRAM Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX480</td>
<td>14</td>
<td>32</td>
<td>48KB</td>
<td>3GB</td>
</tr>
<tr>
<td>GTX285</td>
<td>30</td>
<td>8</td>
<td>16KB</td>
<td>1GB</td>
</tr>
</tbody>
</table>
User writes code for a structured grid calculation

```python
# 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]
```
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 ($256^3$)

```c
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)] = ...;
                }
              }
            }
          }
        }
      }
    }
  }
}
```
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
NVIDIA GTX480 – Varying D

Results – Version Comparison (Raw CUDA)
NVIDIA GTX285 vs. 480

<table>
<thead>
<tr>
<th>GPU</th>
<th>SMs</th>
<th>SIMD</th>
<th>Sh_mem Size</th>
<th>DRAM Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX480</td>
<td>14</td>
<td>32</td>
<td>48KB</td>
<td>3GB</td>
</tr>
<tr>
<td>GTX285</td>
<td>30</td>
<td>8</td>
<td>16KB</td>
<td>1GB</td>
</tr>
</tbody>
</table>
Results – Code Variant Performance

GTX285

<table>
<thead>
<tr>
<th>GPU</th>
<th>SMs</th>
<th>SIMD</th>
<th>Sh_mem Size</th>
<th>DRAM Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX480</td>
<td>14</td>
<td>32</td>
<td>48KB</td>
<td>3GB</td>
</tr>
<tr>
<td>GTX285</td>
<td>30</td>
<td>8</td>
<td>16KB</td>
<td>1GB</td>
</tr>
</tbody>
</table>
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

Example: Speech Recognition

Gaussian Mixture Model for One Phone State

HMM Acoustic Phone Model

Pronunciation Model

Bigram Language Model

Compiled HMM Recognition Network
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.

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

- **Core i7 (45nm)**
- **GTX285 (55nm)**
- 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