Speech Recognition on Vector Architectures

Dissertation: Adam Janin
Presentation: David Sheffield
Content

• If not stated otherwise, the figures are from Adam Janin’s dissertation
• Figures pulled from
  – Jike Chong’s quals slides
  – CS252 slides
Jibbigo demo

Do you accept credit cards?

Akzeptieren sie kreditkarten?

Are you currently taking any medication? (Have you been taking any medication?)

你最近在服什么药吗？(ni3zui4jin4zai4fu2shi2me5yao4 ma3)
Motivation: Speech recognition on mobile devices

- Vector hardware is the key to speech recognition on mobile devices today!
- Let's take a look at the Jibbigo requirements
  - Fast devices have integrated vector units!
  - Slow devices do not have vector units!
- ParLab mobile speech decoder has similar requirements for vector hardware

Apple Device Compatibility Chart:

<table>
<thead>
<tr>
<th>Device</th>
<th>Speed</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone 1 and 3G</td>
<td>slow</td>
<td>unidirectional</td>
</tr>
<tr>
<td>iPhone 3GS and 4</td>
<td>fast</td>
<td>bidirectional</td>
</tr>
<tr>
<td>iPod Touch 1st gen.</td>
<td>no microphone</td>
<td></td>
</tr>
<tr>
<td>iPod Touch 2nd gen.</td>
<td>slow</td>
<td>unidirectional</td>
</tr>
<tr>
<td>iPod Touch 3rd and 4th gen.</td>
<td>fast</td>
<td>bidirectional</td>
</tr>
<tr>
<td>iPad 1 and iPad 2</td>
<td>fast</td>
<td>bidirectional</td>
</tr>
</tbody>
</table>

http://www.jibbigo.com/website/features/jibbigo-iphone
Outline of dissertation

- Introduction
- Vector Simulation Library
- Overview of Speech Recognition
- Signal Processing
- Phone Probability Estimator
- Small Vocabulary Decoder
- Large Vocabulary Decoder
Outline

• Overview of Speech Recognition
  – Signal processing
  – Acoustic likelihood calculation
  – Decoding
• Vector architectures
• Phone probability estimator on vector processors
• Small vocabulary decoder
• Large vocabulary decoder
Speech “vocabulary”

• Phoneme
  – A minimal speech sound which distinguishes two words ("sat vs cat")

• Allophone
  – Allophones are context-dependent variant pronunciations of a phoneme ("sat vs was")

• Language model
  – Provides a score for a given sequence of words based on the likelihood of the sequence for a given language
Speech recognition in a nutshell

- **Voice Input**
- **Signal Processing Module**
- **Speech Features**
- **Inference Engine**
- **Word Sequence**
  - I think therefore I am

**Recognition Network**
- Acoustic Model
- Pronunciation Model
- Language Model

**Signal Sampled**
- 16 bits
- 16 kHz

**Signal Processing**
- Spectral-like Features
  - 20 floats every 10 ms

**Phone Prob. Estimators**
- Neural Network
  - Phone Probabilities
    - /a/ 0.263
    - /e/ 0.104
    - /i/ 0.002
    - ...

**Decoder HMM**
- Recognized Text
  - "the"
  - "car"
  - "is"
The big picture

• Signal processing
  – Feature extraction for later processing steps

• Phone Probability Estimation
  – Estimate the probability that the given features represent a particular sound or combination of sounds

• Decoding
  – Convert sequences of phone probability estimates and compares them against every possible utterance
    • Not tractable in reality, so we prune the search space
Signal processing (1/2)

• MFCC features
  – Pre-emphasis
    • High-pass filter and “mimics some of the equal loudness characteristics of the human auditory system”
  – Windowing
    • Multiply by a Hamming window
  – Filterbank
    • FFT followed element-wise multiplication
  – Logarithm
    • Human perception of loudness is compressive, mimic with logarithm
Signal processing (2/2)

- MFCC features
  - DCT
    - Output from log has high feature-to-feature correlation, use DCT to reduce data to 8 to 14 coefficients
  - MFCC construction
    - 12 MFCC
    - 12 Δ MFCC
    - 12 ΔΔ MFCC
    - 1 energy
    - 1 Δ energy
    - 1 ΔΔ energy
Acoustic Likelihood Computation

• Acoustic likelihood are based on computing observation probabilities directly on the input feature vector
  – Commonly, the 39-dimensional MFCC feature vector

• Janin’s thesis uses neural nets but Gaussian mixture models are common today
  – People have also tried support vector machines and conditional random fields
Multi-layer perceptrons

- Janin uses a two layer neural network
  - Input layer
    - 9 frames of 20 features
    - 2000x180 matrix-vector multiply
    - Sigmoid function
  - Hidden layer
    - 2000 fully connected units
    - 56x2000 matrix-vector multiply
    - Soft-max function
  - Output layer
    - 56 phones
- Evaluation of the MLP is conceptually matrix-vector multiply
  - True of GMMs too
  - Batch processing of input vectors transforms the computation into matrix-vector multiply
    - More later
FSM for “about”

• Conceptually, each word can be thought of as a finite state machine (FSM)

• One state per phone in the word
  – More realistic systems often have several states per phone

• Self-loops encapsulate differences in rate of speech
Speech decoder

• Given phone probabilities from the output of the neural network and finite state machine representing a word in our vocabulary, the job of the decoder is to determine the likelihood that a FSM and phone stream match.
  – Viterbi algorithm used to compute most likely match
    • Dynamic programming approximation for the best match (instead of likelihood of all paths)
An example

- Viterbi dynamic programming for the word “about”
- The variables
  - Time index: t
  - State index: s
  - Table entry: \(E(t, s)\)
  - Likelihood of phone S: \(P(X_t | S_s)\)

\[
E(t, s) = P(X_t | S_s) \cdot \max( E(t-1, s), E(t-1, s-1) )
\]
Speech decoder types

– Small vocabulary decoder
  • Probability of every word in the dictionary is evaluated
  • Becomes inefficient with large language models
  • Algorithms are much more “regular”
    – Regular is very good for vector architectures

– Large vocabulary decoder
  • As the vocabulary becomes larger, more words share the same common prefix
    – Small vocabulary approach duplicates work
  • Rearrange computation so that common prefix decoding occurs only once
Tree structured Lexicons

- Each node represents a phone
  - All words that start with the same sounds are on the same branch
  - Note that the computation starts to be come less regular
Outline

• Overview of Speech Recognition
  – Signal processing
  – Acoustic likelihood calculation
  – Decoding

• Vector architectures

• Phone probability estimator on vector processors

• Small vocabulary decoder

• Large vocabulary decoder
Cray-1: the canonical vector processor

From: http://en.wikipedia.org/wiki/Cray-1
Vector vocabulary

- **SIMD**
  - Single-instruction, multiple data. One instruction operates on multiple operands. It’s a map for you LISP programmers out there.

- **Data-parallel**
  - Roughly, the same thing as SIMD.

- **Vector processor**
  - One possible implementation of a data-parallel processor. My characterization: a single instruction operates on more than 16 elements.
    - X86 have 4 to 8 long SIMD
    - Cray vector machines have 64 long vectors

- **SIMT**
  - Single-instruction, multiple thread. GPUs fall in this category. It really means hardware does masking that a vectorizing compiler would traditionally do.
## Motivating code example

<table>
<thead>
<tr>
<th>C code</th>
<th>Scalar Code</th>
<th>Vector Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;64; i++) C[i] = A[i] + B[i];</td>
<td>LI R4, 64</td>
<td>LI VLR, 64</td>
</tr>
<tr>
<td></td>
<td>loop:</td>
<td>LV V1, R1</td>
</tr>
<tr>
<td></td>
<td>L.D F0, 0(R1)</td>
<td>LV V2, R2</td>
</tr>
<tr>
<td></td>
<td>L.D F2, 0(R2)</td>
<td>ADDV.D V3, V1, V2</td>
</tr>
<tr>
<td></td>
<td>ADD.D F4, F2, F0</td>
<td>SV V3, R3</td>
</tr>
<tr>
<td></td>
<td>S.D F4, 0(R3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DADDIU R1, 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DADDIU R2, 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DADDIU R3, 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSUBIU R4, 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BNEZ R4, loop</td>
<td></td>
</tr>
</tbody>
</table>

---

From CS252 Slides: [http://www.eecs.berkeley.edu/~kubitron/courses/cs252-S07/lectures/lec11-vector.pdf](http://www.eecs.berkeley.edu/~kubitron/courses/cs252-S07/lectures/lec11-vector.pdf)
Benefits of a vector ISA

• Compact
  – One instruction encodes N operations

• Expressive
  – States N operations are independent

• Scalable
  – Can run code on one or more pipelines

From CS252 Slides: http://www.eecs.berkeley.edu/~kubitron/courses/cs252-S07/lectures/lec11-vector.pdf
Scalable pipelines

ADDV C, A, B

Execution using one pipelined functional unit


C[0]
C[1]
C[2]

Execution using four pipelined functional units


A[22] B[22]

A[27] B[27]

C[0]
C[1]
C[2]
C[3]

From CS252 Slides: http://www.eecs.berkeley.edu/~kubitron/courses/cs252-s07/lectures/lec11-vector.pdf
High-level design of a vector processor

Vector Memory-Memory versus Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- The first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
- Cray-1 ('76) was first vector register machine

Example Source Code

```
for (i=0; i<N; i++)
{
    C[i] = A[i] + B[i];
    D[i] = A[i] - B[i];
}
```

Vector Memory-Memory Code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

Vector Register Code

```
From CS252 Slides: http://www.eecs.berkeley.edu/~kubitron/courses/cs252-S07/lectures/lec11-vector.pdf
Matrix Multiply: the killer kernel for vector processors

```c
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++) {
    for (j=1; j<n; j++) {
        for (t=1; t<k; t++)
            sum += a[i][t] * b[t][j];
        c[i][j] = sum;
    }
}
```

```c
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++) {
    for (j=1; j<n; j+=32) {
        for (t=1; t<k; t++) {
            a_scalar = a[i][t]; /* Get scalar */
            b_vector[0:31] = b[t][j:j+31]; /* Get vector */

            /* Do a vector-scalar multiply. */
            prod[0:31] = b_vector[0:31]*a_scalar;

            /* Vector-vector add into results. */
            sum[0:31] += prod[0:31];
        }
        c[i][j:j+31] = sum[0:31];
    }
}
```

From CS252 Slides: http://www.eecs.berkeley.edu/~kubitron/courses/cs252-S07/lectures/lec11-vector.pdf
Berkeley’s vector processors

Torrent-0: ICSI/Berkeley (1995)

VIRAM-1: Berkeley (2002)
Your vector processors (1/2)

• iPhone 3GS or newer?
  – Cortex-A8 with 4-wide SIMD

• x86 PC?
  – Pentium3 through Nehalem (i7)
    • 4-wide SIMD per core
  – Sandy bridge
    • 8-wide SIMD per core

• Game consoles?
  – Xbox360
    • 4-wide SIMD per core (3 processors overall)
  – Playstation3
    • 4-wide SIMD (6 SPUs + 1 PPE)
Your vector processors (II/II)

- Graphics processors (GPUs)
  - Parallel processor comprised of SIMD processors
- For example, the GeForce 580
  - 16 SIMD processors
  - 32-wide SIMD
  - 512 FPUs!
- Lots of work in the ParLab to run speech code on the GPU
Want to know more about vector processors?

- Check out “Computer Architecture: A Quantitative Approach”
  - 5th edition has a chapter comparing GPUs with traditional vector processors
- Take CS252
- Beg Randy Allen to teach vectorizing compilers again
Outline

• Overview of Speech Recognition
  – Signal processing
  – Acoustic likelihood calculation
  – Decoding
• Vector architectures
• Phone probability estimator on vector processors
• Small vocabulary decoder
• Large vocabulary decoder
MLP evaluation and matrix multiply (1/3)

• Why is batch processing of the input features so desirable?
  • Matrix-vector multiply
    – \( O(n^2) \) FPU operations for \( O(n^2) \) operands
  • Matrix-matrix multiply
    – \( O(n^3) \) FPU operands for \( O(n^2) \) operands
    – \( O(n) \) operand reuse!
      » Reuse allows storage of some fraction of the working set in fast memory
        • Register file or cache memory
MLP evaluation and matrix multiply (2/3)

• Stated simply:
  – Matrix-vector multiply performance is proportional to DRAM speed
  – Matrix-matrix multiply performance is proportional to “fast” (cache or register file) speed

• An example on an Nvidia Quadro 6000:
  – Matrix-vector multiply (cublasSgemv)
  – Matrix-matrix multiply (cublasSgemm)
MLP evaluation and matrix multiply (3/3)

• Janin discusses several ways of implementing matrix-multiply
  – His work with vector architecture was tied with the VIRAM research prototype
    • Had to roll his own linear algebra libraries
  – He doesn’t present performance for his different approaches
    • He uses a functional simulation library throughout his thesis
      – Not clear if VIRAM silicon was functional
BLAS and your research

- Modern SIMD processors have vendor provided linear algebra libraries that are really really fast
  - NVIDIA: cublas
  - Intel: Math Kernel Library (MKL)
  - Apple: Accelerate
    - Library is on the iPhone too!
  - AMD: ACML
  - Several open source projects

- FYI: MATLAB’s performance is due to recognition of linear algebra operations then calling BLAS
Small vocabulary decoder on vectors architectures

- We could try to vectorize by column however there would an intra-vector dependences
- Vectorize by row
  - Not particularly efficient if the vector length is much longer than the typical word length

\[ E(t, s) = P(X_t | S_s) \cdot \max(E(t - 1, s), E(t - 1, s - 1)) \]
Small vocabulary decoder on vectors architectures (I/III)

- Extend vector length by placing words side by side
  - We want to pack vector length words words words
    - Example: 16 length vectors want 16 words packing
  - How do we decide what words to place side-by-side?
Outline

• Overview of Speech Recognition
  – Signal processing
  – Acoustic likelihood calculation
  – Decoding

• Vector architectures

• Phone probability estimator on vector processors

• **Small vocabulary decoder**

• **Large vocabulary decoder**
Small vocabulary decoders on vector architectures (II/III)

• N words total in the dictionary
  – Each word has n states
• We want to arrange words into groups that have no more than vector length words
• This is the bin packing problem
  – NP-Complete
    • Really good polynomial (within 22% of optimal) approximations exist (first ordered fit)
Small vocabulary decoder on vector architectures (III/III)

<table>
<thead>
<tr>
<th>Name / States</th>
<th>mv1</th>
<th>Best</th>
<th>Actual</th>
<th>Loss %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digits 97</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Numbers 336</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>14</td>
<td>15</td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Web 941</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>40</td>
<td>41</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>30</td>
<td>31</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>15</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name / States</th>
<th>mv1</th>
<th>Best</th>
<th>Actual</th>
<th>Loss %</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallBN 14793</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>463</td>
<td>470</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>309</td>
<td>314</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>232</td>
<td>235</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>116</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>58</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>MedBN 74717</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>2335</td>
<td>2370</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>1557</td>
<td>1585</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1168</td>
<td>1183</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>584</td>
<td>586</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>292</td>
<td>294</td>
<td>0.7</td>
</tr>
<tr>
<td>LargeBN 485330</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>10112</td>
<td>10273</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>7584</td>
<td>7681</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>3792</td>
<td>3805</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1896</td>
<td>1906</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Outline

• Overview of Speech Recognition
  – Signal processing
  – Acoustic likelihood calculation
  – Decoding

• Vector architectures

• Phone probability estimator on vector processors

• Small vocabulary decoder

• Large vocabulary decoder
Recall: tree structured Lexicons

- Each node represents a phone
  - All words that start with the same sounds are on the same branch
  - Note that the computation starts to be come less regular
Large vocabularies and pruning

• We’ve already avoiding unnecessary prefix computation through tree-structured lexicons
• We can also prune the search space
  – Branch pruning
    • If a prefix is unlikely to match the input stream, then any word starting with the prefix is unlikely to match
  – Phone deactivation pruning
    • If any frame has a phone probability less than a threshold value, its probability is set to zero
      – Example from dissertation: “What is the likelihood that the input stream matches the word four if no frame in the input stream contains /f/ with significant probability?”
Large vocabulary decoders on vector architectures (I/II)

• With large vocabularies, branch and phone deactivation pruning become very effective
  – Significant speed-up on vector architectures required to offset algorithmic benefits of pruned, tree-structured algorithm
  – Brute force only gets you so far....
Large vocabulary decoders on vector architectures (II/II)

<table>
<thead>
<tr>
<th>Dictionary</th>
<th>Size</th>
<th>Vector</th>
<th>Scalar with Tree Structured Lexicon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>No Prune</td>
</tr>
<tr>
<td>Digits</td>
<td>12</td>
<td>97</td>
<td>89 1.1×</td>
</tr>
<tr>
<td>Numbers</td>
<td>30</td>
<td>336</td>
<td>209 1.6×</td>
</tr>
<tr>
<td>Web</td>
<td>79</td>
<td>941</td>
<td>485 1.9×</td>
</tr>
<tr>
<td>SmallBN</td>
<td>1000</td>
<td>14793</td>
<td>9720 1.5×</td>
</tr>
<tr>
<td>MedBN</td>
<td>5000</td>
<td>74717</td>
<td>37347 2.0×</td>
</tr>
<tr>
<td>LargeBN</td>
<td>32010</td>
<td>485330</td>
<td>151901 3.2×</td>
</tr>
</tbody>
</table>
Results with large vocabulary

Speed-up required on a vector architecture to obtain the same performance as an aggressive tree-structured approach.
Parallelization Challenges

- Operations on a graph is driven by the graph structure
  - Irregular structures in a the graph makes it difficult to partition – static load balancing
  - The set of active nodes changes from iteration to iteration – dynamic load balancing
  - Multiple states may share the same next state – write contention in arc evaluation
  - Traversal produces memory accesses that spans the entire graph – poor spatial locality
  - Inner loops have high data access to computation ratio – memory bandwidth limited
Conclusion

• Signal processing and phone probability computation are an excellent match for vector processors
• Small vocabulary decoders work well on vector architectures too
• No effective algorithm for large vocabulary decoders
Conclusions

• Signal processing and phone probability computation are an excellent match for vector processors
• Small vocabulary decoders work well on vector architectures too
• No effective algorithm for large vocabulary decoders
  – Maybe we can solve this today with heterogeneous computing
    • Tightly coupled vector and multicore processors