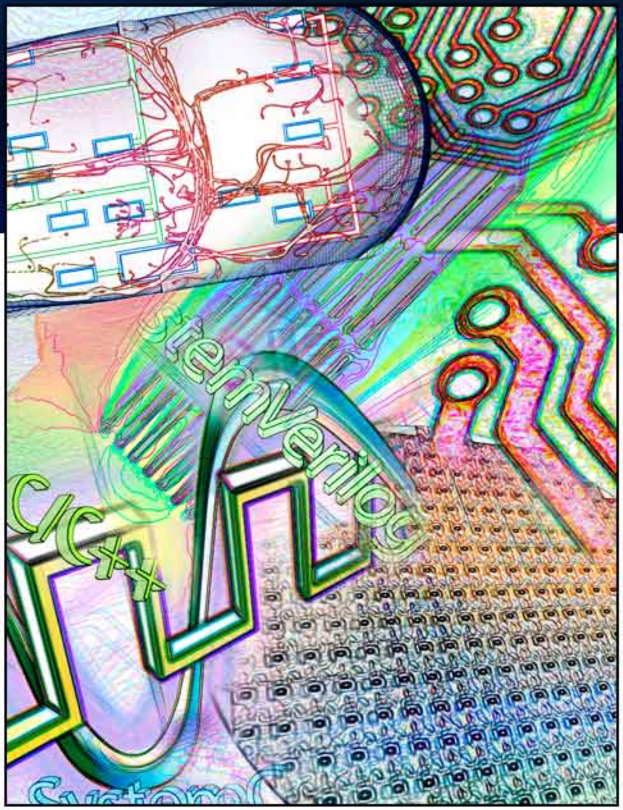
Implementing a Speech Recognition Algorithm with VSIPL++



Don McCoy, Brooks Moses,
Stefan Seefeld, Justin Voo

Software Engineers

Embedded Systems Division / HPC Group

September 2011

CodeSourcery

Mentor Graphics®

Objective

VSIPL++ Standard:

- Standard originally developed under Air Force and MIT/LL leadership.
- CodeSourcery (now Mentor Graphics) has a commercial implementation: Sourcery VSIPL++.
- Intention is to be a general library for signal and image applications.
- Originating community largely focused on radar/sonar.

Question: Is VSIPL++ indeed useful outside of radar and radar-like fields, as intended?

- Sample application: Automatic Speech Recognition

Automatic Speech Recognition

This presentation focuses on two aspects:

- Feature Extraction
- "Decoding" (a.k.a. Recognition)

We will compare to existing Matlab and C code:

- PMTK3 (Matlab modeling toolkit)
  — written by Matt Dunham, Kevin Murphy and others
- MFCC code (Matlab implementation)
  — written by Dan Ellis
- HTK (C-based research implementation)
  — from Cambridge University Engineering Department (CUED)

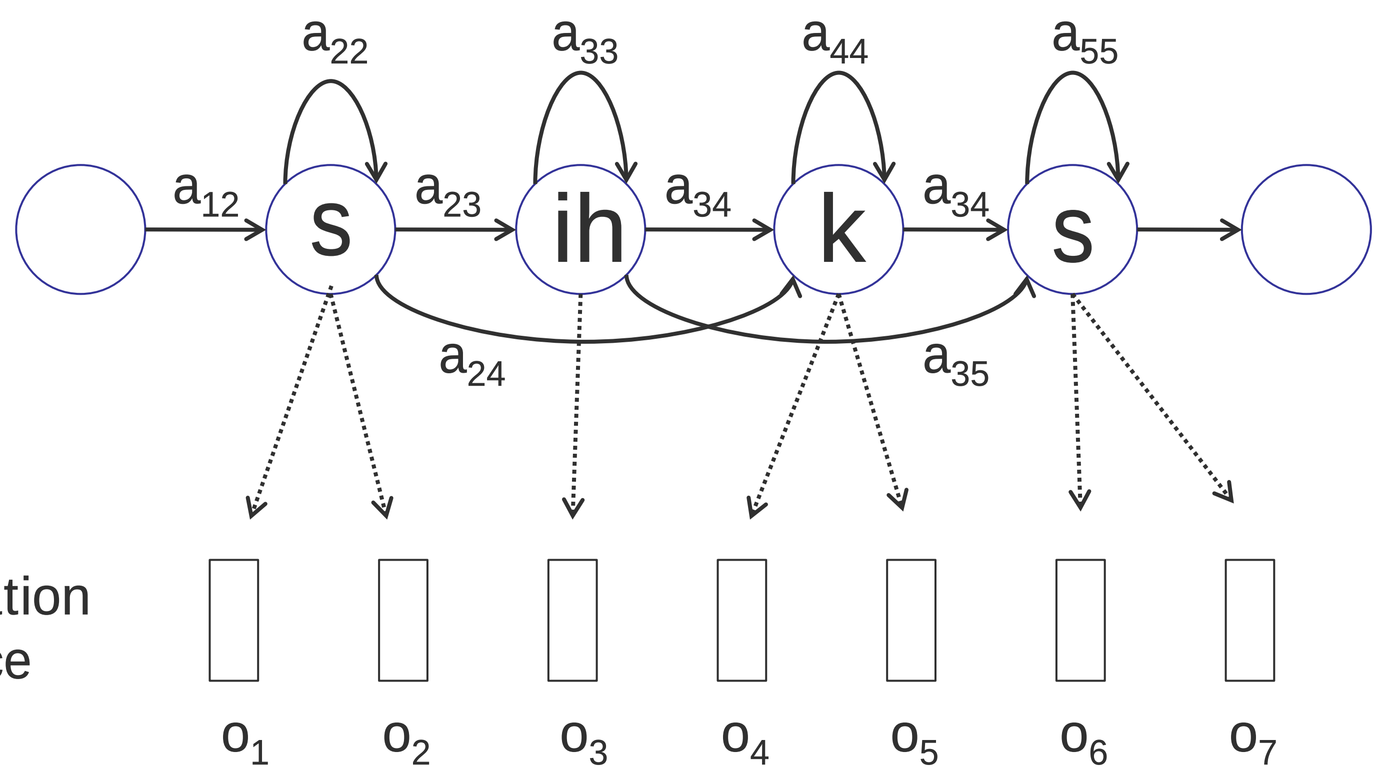
Implementing a Speech Recognition Algorithm with VSIPL++

"DECODING"

Decoding

- Hidden Markov Models (HMM)

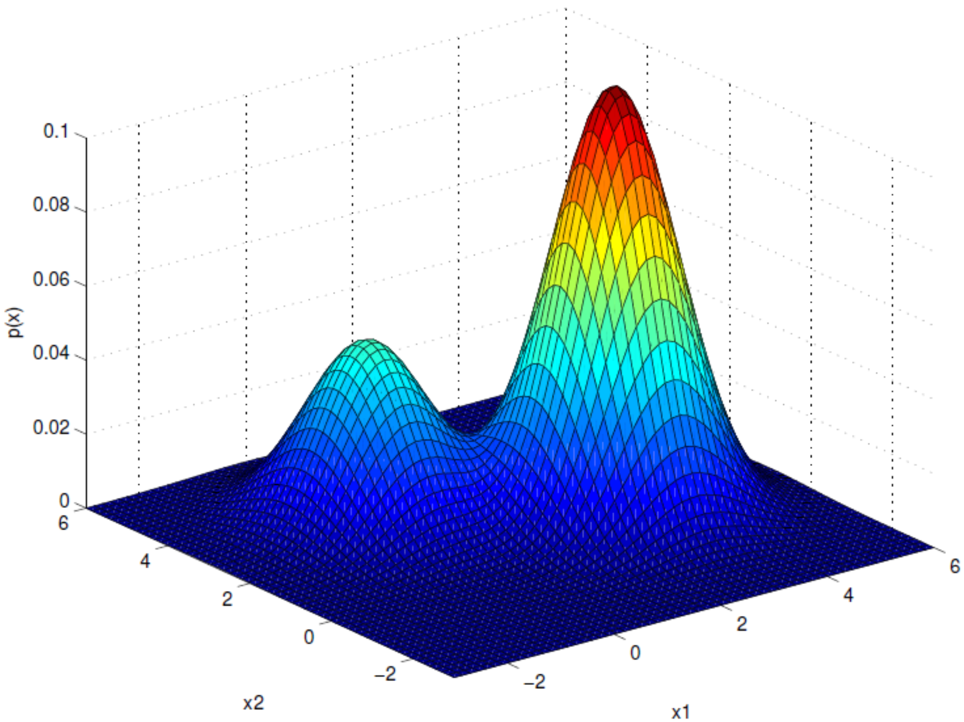
"six"

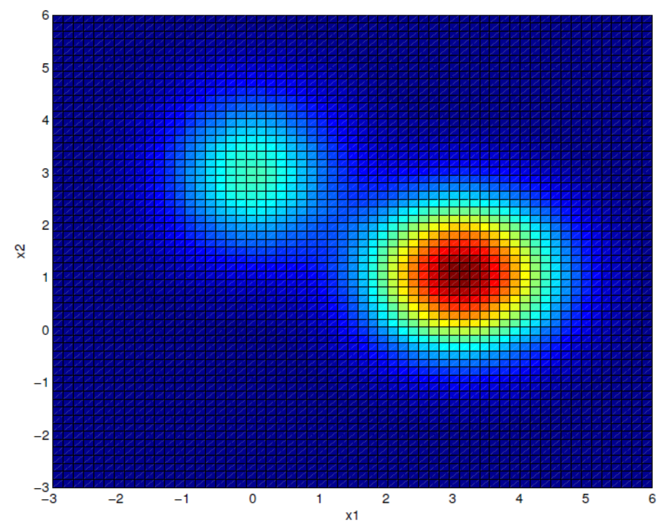


Observation sequence

Decoding

- Gaussian Mixture Models
  - Trained on acoustic data to account for variation
  - Diagram shows representation of PDF for a two-variable MFCC





Decoding

- We wish to find the highest probability word y_n from a sequence of feature vectors $O = \{ o_1, o_2, \ldots o_n \}$, so we maximize (over all words) the *a posteriori* probability

$$P(y_n \mid O)$$

- Using Bayes rule, we find it is sufficient to calculate the log-likelihood

$$\underbrace{\log p(O \mid y_n)}_{\text{Likelihood of a word with a given set of model parameters}} + \log P(y_n)$$

Likelihood of a word
with a given set of
model parameters

Decoding Implementation

- Matlab and VSIPL++ code – PMTK3 hmmFilter() function

```
Matrix<T> AT = transmat.transpose();
alpha = T();
normalize(initDist * softev.col(0), alpha.col(0), scale(0));
for (length_type t = 1; t < tmax; ++t)
  normalize(prod(AT, alpha.col(t-1)) * softev.col(t),
            alpha.col(t), scale(t));
T eps = std::numeric_limits<T>::epsilon();
loglik = sumval(log(scale + eps));
```

```
scale = zeros(T,1);
AT = transmat';
alpha = zeros(K,T);
[alpha(:,1), scale(1)] = normalize(initDist(:) .* softev(:,1));
for t=2:T
  [alpha(:,t), scale(t)] = normalize((AT * alpha(:,t-1)) .*
                                    softev(:,t));
end
loglik = sum(log(scale+eps));
```

Decoding Implementation

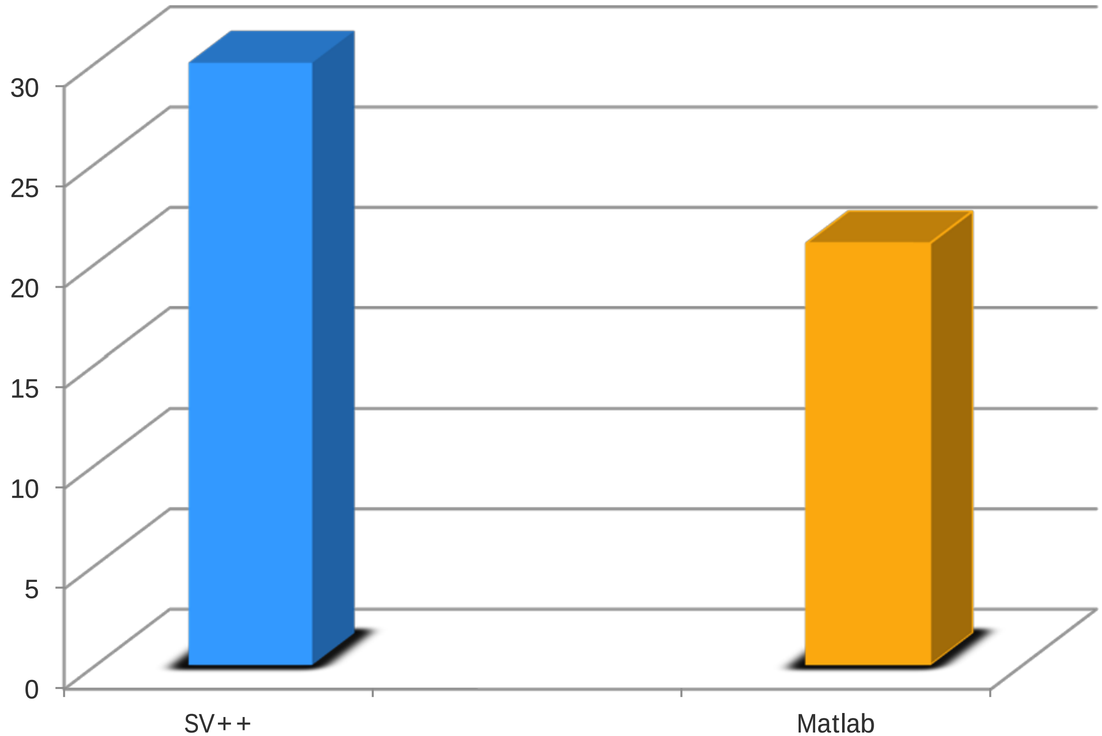
- Matlab and VSIPL++ code – PMTK3 hmmFilter() function

```
Matrix<T> AT = transmat.transpose();
alpha = T();
normalize(initDist * softev.col(0), alpha.col(0), scale(0));
for (length_type t = 1; t < tmax; ++t)
  normalize(prod(AT, alpha.col(t-1)) * softev.col(t),
            alpha.col(t), scale(t));
T eps = std::numeric_limits<T>::epsilon();
loglik = sumval(log(scale + eps));
```

```
scale = zeros(T,1);
AT = transmat';
alpha = zeros(K,T);
[alpha(:,1), scale(1)] = normalize(initDist(:) .* softev(:,1));
for t=2:T
  [alpha(:,t), scale(t)] = normalize((AT * alpha(:,t-1)) .*
                                     softev(:,t));
end
loglik = sum(log(scale+eps));
```

Decoding Implementation

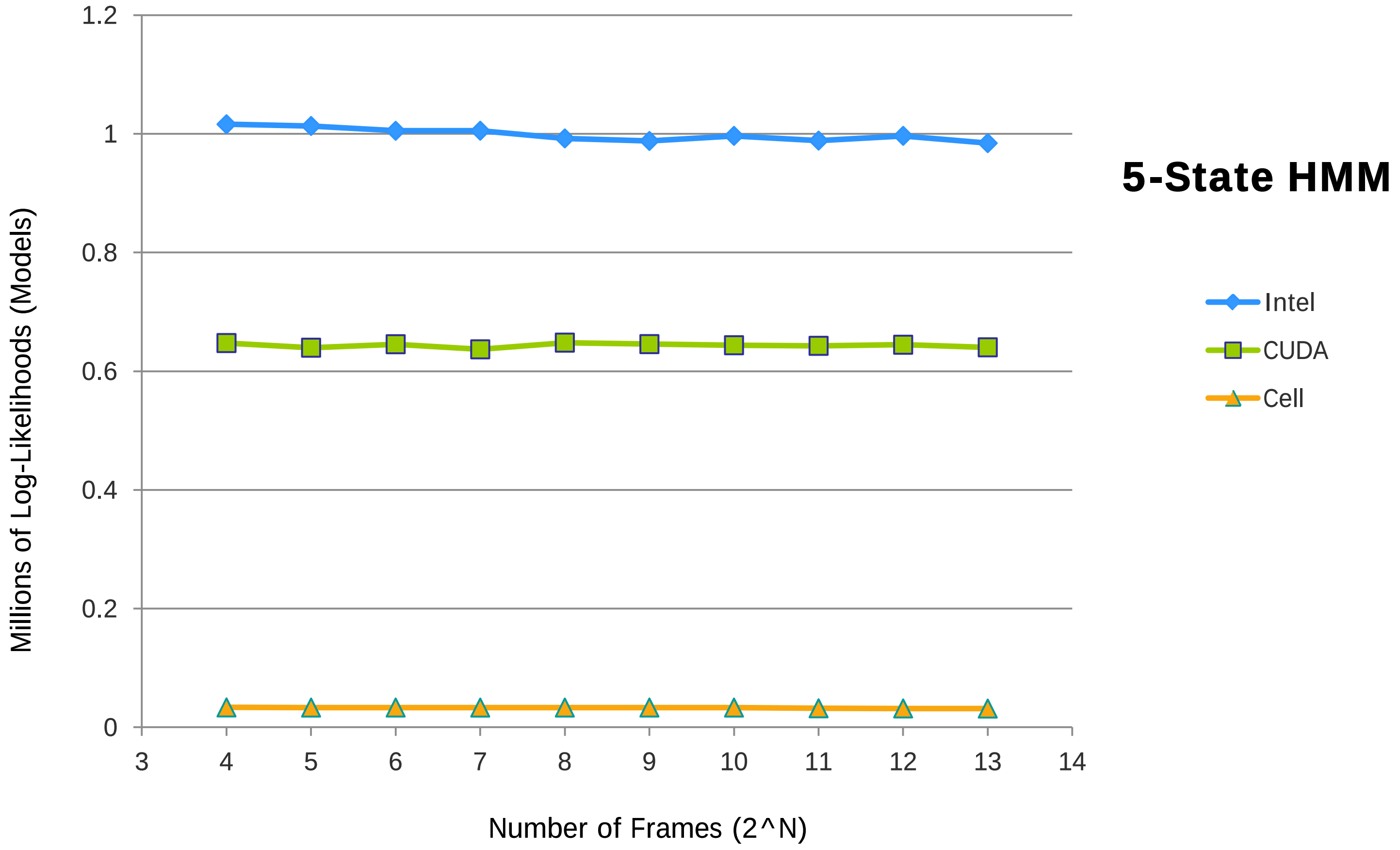
- Similar line counts for both Matlab and VSIPL++
  - VSIPL++: 30
  - Matlab: 21



* generated using David A. Wheeler's 'SLOCCount'.

Decoding Implementation

- Sourcery VSIPL++ Performance



Decoding Implementation

Additional Parallelization Strategies

- Custom kernels (GPU, Cell)
  - Include user-written low-level kernels for key operations
  - Pack more operations into each invocation
  - Take advantage of overlapped computations and data transfers

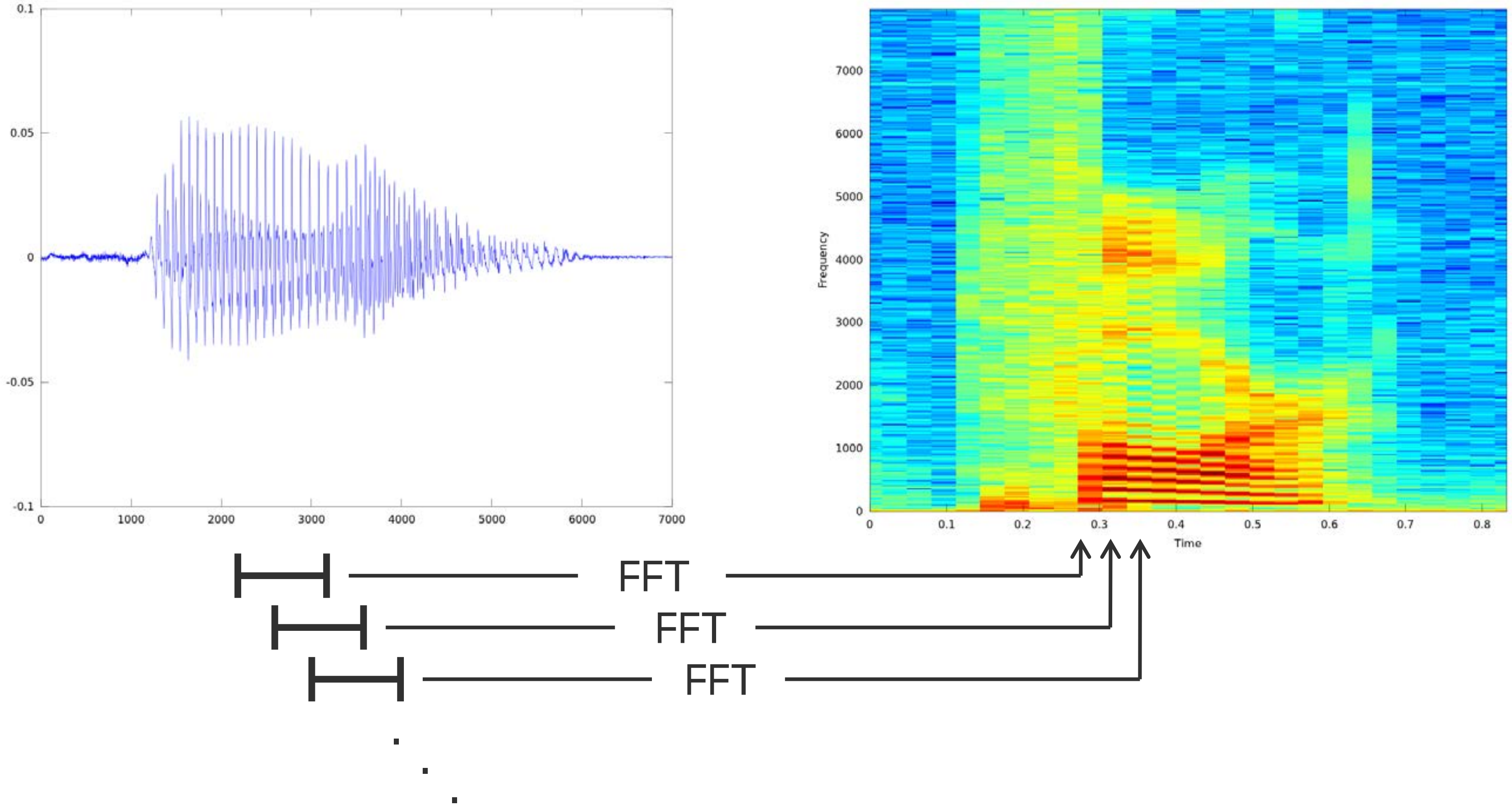
- Maps
  - Distribute VSIPL++ computations across multiple processes
  - Explicit management of multicore/multiprocessor assignment

Implementing a Speech Recognition Algorithm with VSIPL++

FEATURE EXTRACTION

Feature Extraction

- Speech signals – the word "four" and its spectrogram

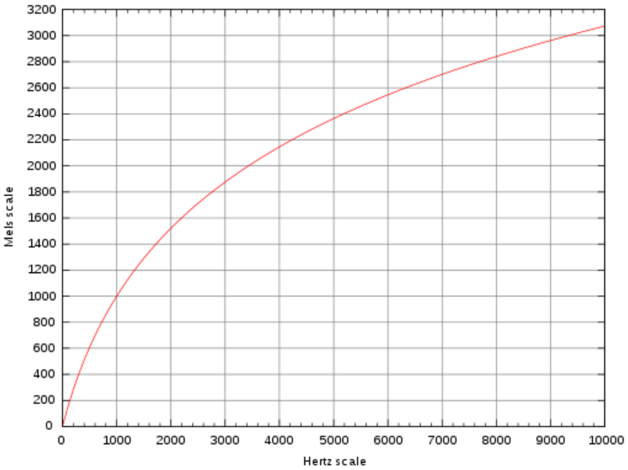


Feature Extraction

- Mel-frequency Cepstral Coefficients
  - 13 coefficients, one of which is overall power across all bands
  - 13 delta coefficients
  - 13 delta-delta, or ‘acceleration’ coefficients
- Result: 39-element “feature” vector for each timeslice
  - A 16 kHz signal chopped into 1024 samples with 50% overlap yields a feature vector every 32 ms.
  - Each second of speech gives about 31 feature vectors.
- Steps:
  - Pre-emphasis
  - FFT
  - Filter bank (spectral warping)
  - DCT

Feature Extraction

- Pre-emphasis
  - High-pass FIR filter: $H(z) = 1 - 09.5z^{-1}$
  - Increases recognition accuracy by leveling the energy present in across frequency bands
- Framing / Windowing / FFT
  - Choose offset between frames (30-50%)
  - Choose frame length (300 – 1024 samples)
- Mel-scale spectral warping
  - Provides compensation for how the auditory system perceives relative differences in pitch
- Discrete Cosine Transform
  - Final product are the cepstral coefficients



Feature Extraction Implementation

- VSIPL++ code

```
// Apply a pre-emphasis filter to the input
Vector<scalar_f> h(2);
h(0) = scalar_f(1);  h(1) = scalar_f(-0.97);
Fir<scalar_f> preemp(h, signal_length);
preemp(x, y);

// Compute the spectrogram of the filtered input
Matrix<std::complex<T> > S = specgram(y,
  hanning(frame_length), frame_offset);

// Integrate into mel bins, in the real domain
Matrix<T> A = prod(wts, magsq(S));

// Convert to cepstra via DCT
Matrix<T> cepstra = spec2cep(A, numceps);
```

Feature Extraction Implementation

- VSIPL++ code

```
// Apply a pre-emphasis filter to the input
Vector<scalar_f> h(2);
h(0) = scalar_f(1);  h(1) = scalar_f(-0.97);
Fir<scalar_f> preemp(h, signal_length);
preemp(x, y);

// Compute the spectrogram of the filtered input
Matrix<std::complex<T> > S = specgram(y,
  hanning(frame_length), frame_offset);

// Integrate into mel bins, in the real domain
Matrix<T> A = prod(wts, magsq(S));

// Convert to cepstra via DCT
Matrix<T> cepstra = spec2cep(A, numceps);
```

Feature Extraction Implementation

- A closer look at the spectrogram function

```
Fft<const_Vector, T, std::complex<T>, 0, by_reference, 1>
  fft(Domain<1>(N), 1.0);

for (length_type m = 0; m < M; ++m)
  fft(in(Domain<1>(m * I, 1, N)) * window, S.col(m));
```

or

```
Fftm<T, complex<T>, col, fft_fwd, by_reference, 1>
  fftm(Domain<2>(N, M), 1.0);
Matrix<T, Dense<2, T, col2_type> > tmp(N, M);

for (length_type m = 0; m < M; ++m)
  tmp.col(m) = in(Domain<1>(m * I, 1, N)) * window;
fftm(tmp, S);
```

Feature Extraction Implementation

- A closer look at the spec2cep function (aka the DCT)

```
template <typename T,
          typename Block>
vsip::Matrix<T>
spec2cep(
  vsip::const_Matrix<T, Block> spec,
  vsip::length_type const ncep)
{
  using namespace vsip;

  Length_type nrow = spec.size(0);
  Matrix<T> dctm(ncep, nrow, T());

  for (length_type i = 0; i < ncep; ++i)
    dctm.row(i) = cos(i * ramp<T>(1, 2, nrow) /
                  (2 * nrow) * M_PI) * sqrt(T(2) / nrow);
  return (prod(dctm, log(spec)));
}
```

Feature Extraction Implementation

- A closer look at the spec2cep function (aka the DCT)

```
template <typename T>
class Dct
{
public:
  Dct(vsip::length_type const ncep, vsip::length_type const nfilts)
    : ncep_(ncep), nfilts_(nfilts), dctm_(ncep, nfilts)
  {
    for (vsip::length_type i = 0; i < ncep; ++i)
      dctm_.row(i) = cos(i * vsip::ramp<T>(1, 2, nfilts) /
                         (2 * nfilts) * M_PI) * sqrt(T(2) / nfilts);
  }
  template <typename Block>
  vsip::Matrix<T>
  operator()(vsip::const_Matrix<T, Block> input)
  {
    return (vsip::prod(dctm_, input));
  }
  ...
};
```

Feature Extraction Implementation

- VSIPL++ using the modified DCT implementation

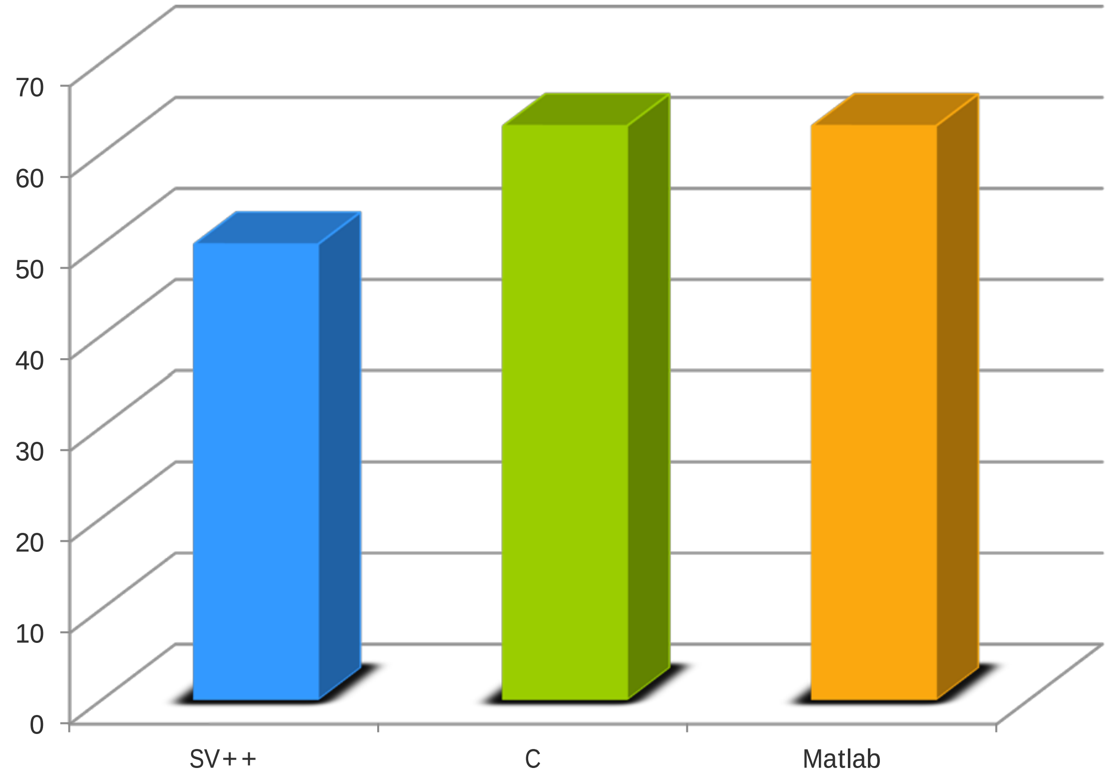
```
Dct<T> dct(numceps, numfilters);

 ...

// Convert to cepstra via DCT
Matrix<T> cepstra = dct(log(A));
```

Feature Extraction Implementation

- Line count comparison*
  - — C++: 63
  - — Matlab: 63
  - — VSIPL++: 50

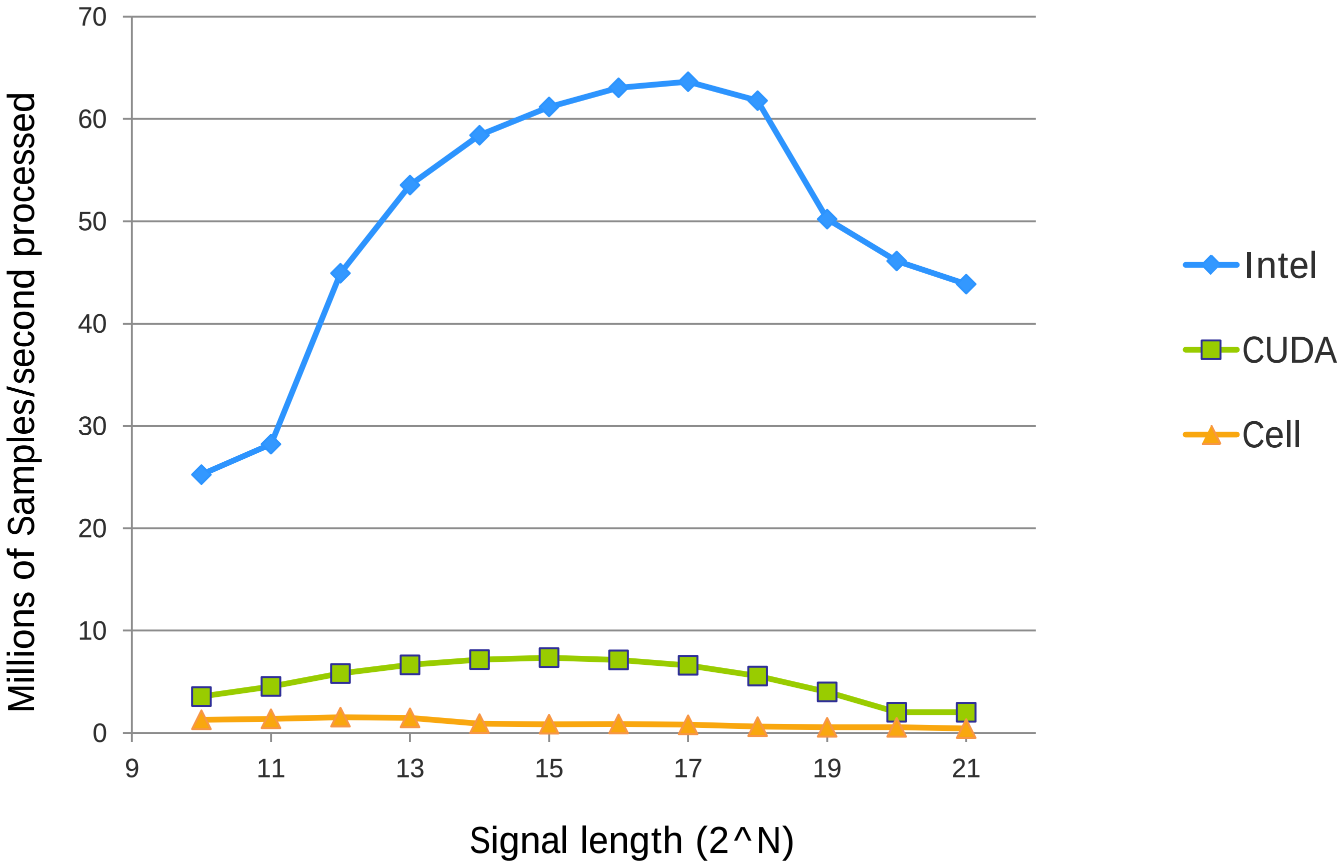


- Performance vs. optimized C (time to process 1000 MFCCs)
  - — C : 1.1 S
  - — Sourcery VSIPL++: 0.8 s

* generated using David A. Wheeler's 'SLOCCount'.

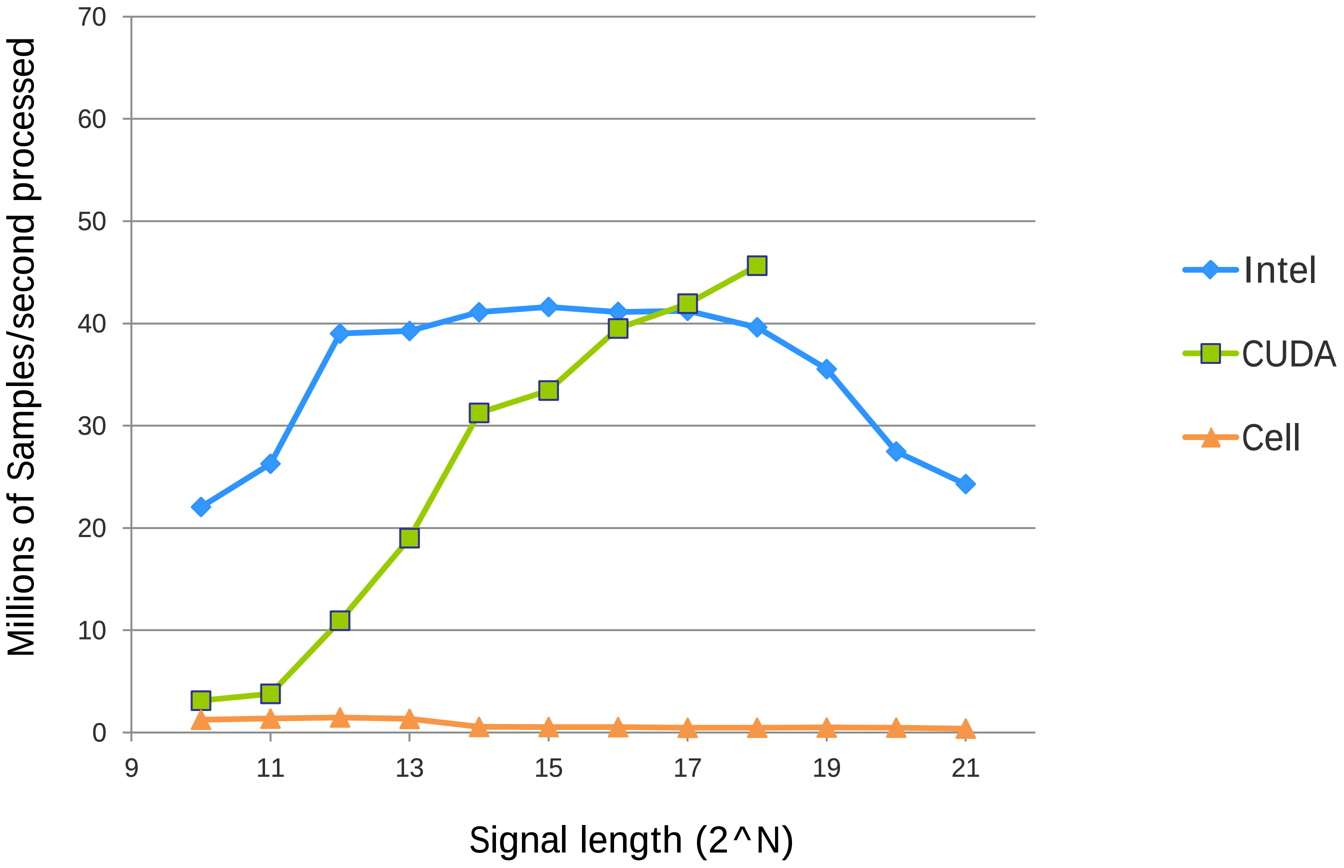
Feature Extraction Implementation

- Sourcery VSIPL++ performance (FFT version)



Feature Extraction Implementation

- Sourcery VSIPL++ performance (multiple-FFT version)



Implementing a Speech Recognition Algorithm with VSIPL++

CONCLUSIONS

VSIPL++ Assessment & Conclusions

Benefits of the VSIPL++ Standard

- Does not tie the user's hands with regard to algorithmic choices
  - Prototyping algorithms is fast and efficient for users
- C++ code is easier to read and more compact
  - Fosters rapid development
- Implementers of the standard have the flexibility required to get good performance.
  - Allows best performance on a range of hardware
- Prototype code is benchmark-ready...

VSIPL++ Assessment & Conclusions, cont…

Potential Extensions to the VSIPL++ Standard

- Direct Data Access (DDA)
  - Already proposed to standards body
  - Proven useful in the field
- Sliding-window FFT / FFTM
- DCT and other transforms

Mentor Graphics®

www.mentor.com

References

- PMTK3
  - probabilistic modeling toolkit for Matlab/Octave, version 3
  - by Matt Dunham, Kevin Murphy, et.al.
  - http://code.google.com/p/pmtk3/
- PLP and RASTA (and MFCC, and inversion) in Matlab
  - by Daniel P. W. Ellis, 2005
  - http://www.ee.columbia.edu/~dpwe/resources/matlab/rastamat/
- HTK
  - Hidden Markov Model Toolkit (HTK)
  - by the Machine Intelligence Laboratory at Cambridge University
  - http://htk.eng.cam.ac.uk/
- Sourcery VSIPL++
  - Optimized implementation of the VSIPL++ standard
  - http://www.mentor.com/embedded-software/codesourcery