CELT: A Low-latency, High-quality Audio Codec

Dr. Jean-Marc Valin, Gregory Maxwell, and Dr. Timothy B. Terriberry
Outline

- Introduction and Motivation
- CELT Design
- libcelt
- Demo
- Conclusion
Introduction

- Two common types of lossy audio codecs
  - Speech/communication (G.72x, GSM, AMR, Speex)
    - Low delay (15-30 ms)
    - Low sampling rate (8 kHz to 16 kHz): limited fidelity
    - No support for music
  - General purpose (MP3, AAC, Vorbis)
    - High delay (> 100 ms)
    - High sampling rates (44.1 kHz or higher)
    - "CD-quality" music
  - We want both: high fidelity with very low delay
Introduction

- Low delay is critical to live interaction
  - Prevents collisions during conversation
  - Reduce need for echo cancellation
    - Good for small, embedded devices without much CPU
  - Higher sense of presence
- Allows synchronization for live music
  - Need less than 25 ms total delay to synchronize (Carôt 2006)
  - Equivalent to sitting 8 m apart (farther requires a conductor)
- Lower delay in the codec increases range
  - 1 ms = 200 km in fiber

High delay (~250 ms)  Low delay (~15 ms)
Introduction

• No entrenched standard in this space
  – G.722.1C (ITU-T) [40 ms delay, up to 32 kHz]
  – AAC-LD (MPEG) [20-50 ms delay, up to 48 kHz]
  – ULD (Fraunhofer) [< 10 ms delay, up to 48 kHz]

• CELT is already ahead of the competition
  – Delay: Configurable, 1.3 ms to 24 ms, ~8 ms typical
  – Quality (at equivalent rates): Much better than G.722.1C, as good as or better than AAC-LD, better than ULD
  – Flexibility: 24 kbps to 160+ kbps, 32 kHz to 96 kHz, configurable delay, low-complexity mode
  – Freedom: Open source (BSD), no patents
Outline

- Introduction
- CELT Design
- libcelt
- Demo
- Conclusion
CELT: "Constrained Energy Lapped Transform"

- Transform codec (MDCT, like MP3, Vorbis)
  - Short windows (~8 ms) → poor frequency resolution
- *Explicitly* code energy of each band of the signal
  - Coarse shape of sound preserved no matter what
- Code remaining details using vector quantization
- Also uses pitch prediction with a time offset
  - Similar to linear prediction used by speech codecs
  - Helps compensate for poor frequency resolution
Outline

- Introduction
- CELT Design
  - "Lapped Transform"
  - "Constrained Energy"
  - Coding Band Shape
  - Performance Tests
- libcelt
- Demo
- Conclusion
"Lapped Transform"
Time-Frequency Duality

- Any signal can be represented as a weighted sum of cosine curves with different frequencies.
- The Discrete Cosine Transform (DCT) computes the weights for each frequency.

![Graph showing the DCT of a signal with frequencies 220 Hz (A3), 440 Hz (A4), and 1245 Hz (D#5).](chart)
"Lapped Transform"
Discrete Cosine Transform

- The "Discrete" in DCT means we're restricted to a finite number of frequencies
  - As the transform size gets smaller, energy "leaks" into nearby frequencies (harder to compress)

N=48000
"Lapped Transform"
Discrete Cosine Transform

- The "Discrete" in DCT means we're restricted to a finite number of frequencies
  - As the transform size gets smaller, energy "leaks" into nearby frequencies (harder to compress)

N=4096 (Maximum Vorbis block size)
The "Discrete" in DCT means we're restricted to a finite number of frequencies

- As the transform size gets smaller, energy "leaks" into nearby frequencies (harder to compress)

N=1024 (Typical Vorbis block size)
"Lapped Transform"
Discrete Cosine Transform

- The "Discrete" in DCT means we're restricted to a finite number of frequencies
  - As the transform size gets smaller, energy "leaks" into nearby frequencies (harder to compress)

N=256 (CELT can use 64...512)
"Lapped Transform"
Discrete Cosine Transform

- The "Discrete" in DCT means we're restricted to a finite number of frequencies
  - As the transform size gets smaller, energy "leaks" into nearby frequencies (unstable over time)

N=256 (CELT can use 64...512)
Frame 2...
"Lapped Transform"
Discrete Cosine Transform

- The "Discrete" in DCT means we're restricted to a finite number of frequencies
  - As the transform size gets smaller, energy "leaks" into nearby frequencies (unstable over time)

N=256 (CELT can use 64...512)
Frame 3...
"Lapped Transform"
Modified DCT

- The normal DCT causes coding artifacts (sharp discontinuities) between blocks, easily audible
- The "Modified" DCT (MDCT) uses a decaying window to overlap multiple blocks
  - Same transform used in MP3, Vorbis, AAC, etc.
  - But with much smaller blocks, less overlap
Outline

- Introduction
- CELT Design
  - "Lapped Transform"
  - "Constrained Energy"
  - Coding Band Shape
  - Performance Tests
- libcelt
- Demo
- Conclusion
"Constrained Energy"

Critical Bands

- The human ear can hear about 25 distinct "critical bands" in the frequency domain
  - Psychoacoustic masking within a band is much stronger than between bands


Threshold of detection in the presence of masker at 1kHz with a bandwidth of 1 critical band and various levels.
"Constrained Energy"

Critical Bands

- Group MDCT coefficients into bands approximating the critical bands (Bark scale)
  - We limit bands to contain at least 3 coefficients to minimize per-band overhead
  - Insufficient frequency resolution for all the bands
  - But we spend most of our bits on LFs anyway
"Constrained Energy" Coding Band Energy

- Most important psychoacoustic lesson learned from Vorbis:
  
  *Preserve the energy in each band*

- Vorbis does this implicitly with its "floor curve"

- CELT codes the energy explicitly
  - Coarse energy (6 dB resolution), predicted from previous frame and from previous band
    - Prediction saves 28 bits/frame, 5.6 kbps with 5 ms frames
  - Fine energy, improves resolution where we have available bits, not predicted
"Constrained Energy"
Coding Band Energy

- CELT (green) vs original (red)
  - Notice the quantization between 8.5 kHz and 12 kHz
  - Speech is intelligible using coarse energy alone (~9 kbps for 5.3 ms frame sizes) 🎤
Outline

- Introduction
- CELT Design
  - "Lapped Transform"
  - "Constrained Energy"
  - **Coding Band Shape**
  - Performance Results
- libcelt
- Demo
- Conclusion
Coding Band Shape

• After normalizing, each band is represented by an $N$-dimensional unit vector
  – Point on an $N$-dimensional sphere
  – Describes "shape" of energy within the band

• Code this shape using two pieces:
  – An *adaptive codebook* using previously decoded signal content to predict the current frame
  – A *fixed codebook* to handle the part of the signal that can't be predicted (the "innovation")

• Latter uses *vector quantization*
Coding Band Shape
Vector Quantization

- Approximates a multidimensional distribution with a finite number of codewords (vectors)

Scalar Quantization (2 bits/dim)

Vector Quantization (2 bits/dim)

RMS error = 0.89

RMS error = 0.71
(20% better)
Coding Band Shape
Vector Quantization

- Easily scales to less than 1 bit per dimension (very important for HF bands: 50-200 dims)

Scalar Quantization (0.5 bits/dim)

Vector Quantization (0.5 bits/dim)

RMS error = 2.93

RMS error = 1.63
(44% better)
• CELT requires a *lot* of codebooks
  − Every band can have a different # of dimensions
  − Exact number of bits available for each band varies from packet to packet

• CELT requires *large* codebooks
  − Exponential in # of dimensions: 50 dims at 0.6 bits/dim. requires over a billion codebook entries
  − We couldn't even store one codebook that large
  − And even if we could, it'd take forever to search

• But we have uniformly distributed unit vectors
Use a regularly structured, algebraic codebook: Pyramid Vector Quantization (Fischer, 1986)

- We want evenly distributed points on a sphere
  - Don't know how to do that for arbitrary dimension
  - Use evenly distributed points on a pyramid instead

- For $N$ dimensional vector, allocate $K$ "pulses"

- Codebook consists vectors with integer coordinates whose magnitudes sum to $K$

$$S(N, K) = \{ y \in \mathbb{Z}^N : \sum_{i=1}^{N} |y_i| = K \}$$
Coding Band Shape
Pyramid Vector Quantization

- PVQ codebook has a fast enumeration algorithm
  - Converts between vector and integer codebook index
  - $O(N+K)$ (lookup table, muls) or simpler $O(NK)$ (adds)
  - Latter great for embedded processors, often faster
- Fast codebook search algorithm: $O(N \cdot \min(N,K))$
  - Divide by $L_1$ norm, round down: at least $K-N$ pulses
  - Place remaining pulses (at most $N$) one at a time
- Codebooks larger than 32 bits
  - Split the vector in half and code each half separately
Coding Band Shape
Pitch Prediction

- Short block sizes → poor frequency resolution
  - Speech/music have periodic, stationary content
  - Can't represent the period accurately via short MDCT
- Pitch prediction compensates for poor resolution
  - Search the past 1024 decoded samples in the time domain, code the offset of the best match
    - Resolution equal to the sampling rate
    - Range (48 kHz, FS=256): $\frac{48000}{1024}$ to $\frac{48000}{384} = 46.875$ Hz to 125 Hz
  - Apply an MDCT to convert to the freq. domain
  - Confine prediction to bands below 8kHz
Coding Band Shape Mixing

- Scale each band of pitch MDCT to unit norm: \( p \)
- Compute a pitch gain, \( g_a \in [0...1] \) for each band
- Mix with the fixed codebook vector \( y \) via
  \[
  \tilde{x} \triangleq \tilde{g}_a p + g_f y
  \]
- Output must have unit norm, so \( g_f \) is completely determined:
  \[
  g_f \triangleq \frac{\sqrt{\tilde{g}_a^2 (y^T p)^2 + y^T y (1 - \tilde{g}_a^2)} - \tilde{g}_a y^T p}{y^T y}
  \]
Coding Band Shape
Adaptive vs. Fixed Codebooks

Before applying pitch prediction

- 5.25 bits
- 6.04 bits
- 7.19 bits
- 8.01 bits

After applying pitch prediction

- Tried stronger adaptation, but required more CPU for no perceptible gain
Coding Band Shape Mixing

- Ideal $g_a$ chosen so that residual $r = x - g_a p$ orthogonal to $p$

- Quantizing $g_a$ means orthogonality not exact
  - Used to use basic VQ to code all $g_a$ values at once
  - Now use 1 bit per band, $g_a$ is either 0 or 0.9
Rate Allocation

• Only CBR supported
  – VBR requires buffering, and buffering means delay
  – User specifies the exact number of bytes to encode each packet into
  – Can change from packet to packet, to adapt to channel statistics

• Only a few things are variable-sized
  – Coarse energy (entropy coded)
  – Pitch parameters (can be omitted if not useful)
  – PVQ codewords over 32 bits (rare)
Rate Allocation

- Each band's share of available bits is **fixed**
  - CELT transmits *no* side information for allocation
  - Equivalent to modeling within-band masking
    - "Signal-to-mask" ratio for each band is roughly constant
  - Ignores inter-band masking and tone vs. noise effects
Psychoacoustic Tricks

- Avoiding "birdie" artifacts
  - $K$ may be small, giving a sparse spectrum $> 8$ kHz
  - Use *spectral folding*, a scaled copy of lower-frequency MDCT coefficients, in place of $p$
    - Acts as a cheap source of time-localized noise
    - Mix using a small value for $g_a$ (a function of $K$)

- Avoiding "pre-echo" artifacts
  - When a strong transient is detected, split the frame and do a smaller MDCT on each piece
  - Interleave the results and continue as normal
Block Diagram

Disabled in low-complexity mode
Future Work

• Freeze bitstream format
  – No side information for allocation means *many* details of the encoding become normative

• Dynamic rate allocation
  – Hard to do psychoacoustic analysis without delay
  – Almost any per-band overhead uses a lot of bits

• Improve stereo coupling
  – Currently using PVQ to handle phase vs. magnitude

• Improve pitch prediction
Outline

● Introduction
● CELT Design
  – "Lapped Transform"
  – "Constrained Energy"
  – Coding Band Shape
  – **Performance Results**
● libcelt
● Demo
● Conclusion
CELT vs. The Competition

- Results from Dr. Christian Hoene for ITU-T Workshop last September
CELT vs. The Competition

- Results from Dr. Christian Hoene for ITU-T Workshop last September

![Graph showing PEAQ Score (ODG) for different bitrates and codecs.

- X-axis: Codec types (AAC20ms, CELT 8.7ms, MP3 100ms, ULD 5.4ms)
- Y-axis: PEAQ Score (ODG)
- Color codes:
  - Bitrate 48000
  - Bitrate 64000
  - Bitrate 96000

The Xiph.Org Foundation
Quality vs. Delay (v0.5, no pitch)
Listening Tests – 48 kbps (v0.3.2, with pitch)
Listening Tests – LC Mode (v0.5, no pitch)
Packet Loss
Bit Errors vs. Position

- Wireless transmission means individual bits can be corrupted without causing packet loss
  - Quality loss due to bit errors varies with location in a packet
  - Trellis Coded Modulation (TCM) can give better protection to earlier bits
Example

- Original file (706 kbps)
- Scalar Quantization (227 kbps, SNR=20.9 dB)
  - 5.15 bits per sample
- Encoded with CELT (64.8 kbps, SNR=20.9 dB)
  - 1.47 bits per sample (Frame Size=256)
- Scalar Quantization Residual (amplified 2×)
- CELT Residual (amplified 2×)
  - Throw away information only where it's masked by something else in the signal
Outline

- Introduction
- CELT Design
- libcelt
- Demo
- Conclusion
libcelt

- **Extremely** light-weight fixed-point impl.

<table>
<thead>
<tr>
<th></th>
<th><strong>Full CELT</strong></th>
<th><strong>LC mode</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Enc/Dec State (each)</td>
<td>4.5 kB</td>
<td>0.5 kB</td>
</tr>
<tr>
<td>Required Stack</td>
<td>11-13 kB</td>
<td>7 kB</td>
</tr>
<tr>
<td>Table Data (ROM)</td>
<td>5.5 kB</td>
<td>5.5 kB</td>
</tr>
<tr>
<td>CPU (TI-C55x DSP)</td>
<td>60 MIPS (enc)+</td>
<td>~30 MIPS (enc)+</td>
</tr>
<tr>
<td></td>
<td>30 MIPS (dec)</td>
<td>~15 MIPS (dec)</td>
</tr>
</tbody>
</table>

- Also has a floating-point implementation
  - Requires twice the RAM for CELT-LC, an extra 0.5 kB for full CELT.
  - 0.9% of one core on a 3 GHz Core2
Outline

- Introduction
- CELT Design
- libcelt
  - The API
  - Low-latency Linux Audio
- Demo
- Conclusion
libcelt API

CELTMode
*celt_mode_create(celt_int32_t Fs, int channels, int frame_size,
    int *error);

int
  celt_mode_info(const CELTMode *mode, int request,
                 celt_int32_t *value);
  ● CELT_GET_FRAME_SIZE, CELT_GET_LOOKAHEAD,
    CELT_GET_NB_CHANNELS, CELT_GET_BITSTREAM_VERSION

CELTEncoder
*celt_encoder_create(const CELTMode *mode);

int
  celt_encoder_ctl(CELTEncoder *st, int request,...);
  ● CELT_SET_COMPLEXITY_REQUEST, CELT_SET_COMPLEXITY(x) /*0-10 (int)*/
  ● CELT_SET_LTP_REQUEST, CELT_SET_LTP(x) /*0 or 1 (int)*/

int
  celt_encode(CELTEncoder *st, const celt_int16_t *pcm,
              celt_int16_t *optional_synthesis,
              unsigned char *compressedBytes, int nbCompressedBytes);

void
  celt_encoder_destroy(CELTEncoder *st);

CELTDecoder
*celt_decoder_create(const CELTMode *mode);

int
  celt_decode(CELTDecoder *st, unsigned char *compressedBytes,
              int nbCompressedBytes, celt_int16_t *pcm);

void
  celt_decoder_destroy(CELTDecoder *st);

void
  celt_mode_destroy(CELTMode *mode);
Hello Encoder

```c
#include <stdio.h>
#include <stdlib.h>
#include <celt/celt.h>

int main(int argc, const char *argv[])
{
    celt_int16_t    in[256];
    unsigned char   out[43];
    CELTMode       *mode;
    CELTEncoder    *enc;
    mode=celt_mode_create(48000,1,256,NULL);
    if(mode==NULL) return EXIT_FAILURE;
    enc=celt_encoder_create(mode);
    if(enc==NULL) return EXIT_FAILURE;
    while(fread(in,sizeof(celt_int16_t),256,stdin)>=256){
        if(celt_encode(enc,in,NULL,out,43)<0) return EXIT_FAILURE;
        fwrite(out,sizeof(unsigned char),43,stdout);
    }
    celt_encoder_destroy(enc);
    celt_mode_destroy(mode);
    return EXIT_SUCCESS;
}
```
Hello Decoder

#include <stdio.h>
#include <stdlib.h>
#include <celt/celt.h>

int main(int argc, const char *argv[]){
    unsigned char in[43];
    celt_int16_t out[256];
    CELTMode *mode;
    CELTDecoder *dec;
    celt_int32_t skip;
    mode = celt_mode_create(48000, 1, 256, NULL);
    if (mode == NULL) return EXIT_FAILURE;
    celt_mode_info(mode, CELT_GET_LOOKAHEAD, &skip);
    dec = celt_decoder_create(mode);
    if (dec == NULL) return EXIT_FAILURE;
    while (fread(in, sizeof(unsigned char), 43, stdin) >= 43){
        if (celt_decode(dec, in, 43, out) < 0) return EXIT_FAILURE;
        fwrite(out + skip, sizeof(celt_int16_t), 256 - skip, stdout);
        skip = 0;
    }
    celt_decoder_destroy(dec);
    celt_mode_destroy(mode);
    return EXIT_SUCCESS;
}
Outline

● Introduction
● CELT Design
● libcelt
  – The API
  – *Low-latency Linux Audio*
● Demo
● Conclusion
Low-latency Linux Audio

- Audio hardware often doesn't work with small buffer sizes
  - 256 samples (5.3 ms) sometimes fails
  - Even 512 samples (10.6 ms) occasionally fails
  - I don't know how often this is a Linux driver problem vs. a hardware problem, but...

- There's no easy way to tell if it will work other than to try it and fail
  - And this is Linux's problem
Low-latency Linux Audio

- Even if small buffers work, scheduling delays can prevent us from filling them on time
  - Loading/unloading drivers still causes huge delays, even with RT patches
    - Hot-plugging some USB devices virtually guarantees deadline miss
- Network latency is also critical
  - Some drivers will attempt to throttle interrupts when sending hundreds of packets a second
    - This only makes latency worse
  - Some wi-fi drivers have weird spikes over 100ms (OpenMoko FreeRunner)
Low-latency Linux Audio

• Library support also important
  – On x86-64, glibc's exp() takes substantially longer than average for some arguments
    • Turns out it uses a generic C implementation
    • Includes its own custom multi-precision arithmetic library to compute hundreds of digits of intermediate results if necessary so that the rounding is exactly right
  – expf() is even slower than exp()
    • Changes exception handling mode of FPU, even if it's already set correctly, then changes it "back"

• Now imagine all the dependencies of a video-conferencing app...
Outline

- Introduction
- CELT Design
- libcelt
- **Demo**
- Conclusion
Outline

- Introduction
- CELT Design
- libcelt
- Demo
- **Conclusion**
Conclusion

- CELT brings CD-quality sound to VoIP-style low-delay applications
  - Better than MP3 and <10 ms delay
- Better than emerging proprietary standards
  - As good or better than AAC-LD with half the delay
  - Better quality and error robustness than ULD
  - Supports wider range of bitrates, sampling rates
Early Adopters

• CELT is already being used by a number of projects
  – Soundjack (Alexander Carôt)
    http://virtualsoundexchange.net/node/21
  – NexGenVoIP (Dr. Christian Hoene)
    http://www.nexgenvoip.org/
  – FreeSWITCH (Anthony Minesale II, Brian K. West)
    http://www.freeswitch.org/ (source code available)
  – jack-audio-connection-kit (netjack) (Torben Hohn)
    http://jackaudio.org/ (source code available)
    http://www.radiochnc.com/
Questions?