Multi-threaded Opus and AAC encoding

Introducing SuperFast codec drivers for the fre:ac audio converter

I always liked doing performance optimizations – such projects are often fun to work at and it's very rewarding when an idea actually works out in practice.

tl;dr: *I implemented multi-threaded drivers for Opus Audio as well as the FAAC and Apple Core Audio AAC codecs – more to come. Scroll down or click here to download experimental fre:ac builds with this technology.*

The biggest step for the fre:ac project regarding performance was the introduction of a multi-threaded conversion engine in 2014. Running multiple conversions at the same time allowed fre:ac to scale almost linearly with the number of CPU cores, compared to converting only one file at a time.

Serial vs. parallel conversion in fre:ac

Parallel conversions, however, only help when there actually are multiple items to convert in the first place. When converting less files than there are CPU cores available, it's still inefficient. And when combining multiple files into a single output file, there is still only one CPU core used.

Converting less files than there are CPU cores and converting to a single output file

So, while a big step forward, the situation was still a little unsatisfying in certain cases which made me explore ways to further optimize the conversion engine.

Choosing the best method

In general, there are three different ways to distribute the work done by an audio codec to multiple CPU cores:

1. Low level data parallelism – parallelizing loops
   To make use of low level data parallelism, you look for loops that do the same operation on many data values. A basic example would be multiplying a large array of values with a constant expression. You can spread the work over multiple threads, each doing the

operation on a part of the values. For such optimization, you would usually use something like OpenMP which makes loop parallelization pretty easy. When it comes to complex algorithms, though, not all loops will be parallelizable, limiting the maximum speed-up you can gain with this method. An example of this approach are the Lancer optimizations for the Vorbis audio codec.

2. Task parallelism – pipelining subtasks
   A second method is pipelining. It can be used if the work to be done consists of multiple stages. For an audio codec, these stages might be MDCT processing (converting time-domain samples to frequency values), psychoacoustic analysis and actual audio encoding. While the later stages depend on the results of the earlier ones, audio codecs usually work on blocks of samples, so a later stage can start its work as soon as the preceding stage finished a block. However, this approach only scales well as long as you have enough compute-intensive stages to fully utilize the available CPU cores. It can be combined with loop parallelization, though, to achieve further speedup. LAME MT is an example of a project using this method (without additional loop parallelization).
3. High level data parallelism – splitting work into fragments
   The third method splits the work to be done into multiple parts each processed by a separate thread. For example, the first and second half of an audio file can be encoded separately and the results later be joined back together. Seeming very easy at first glance, this method has some issues due to frames in an audio file not being completely independent. The LAME MT project tried this method at first, but later found that the project's constraints of producing a result bit-identical to the non-parallel version could not be met with this approach and resorted to pipelining.

Looking at and thinking about the three methods, I quickly came to the conclusion that the first two were not feasible for a project like fre:ac. They would require modifications deep in the respective codecs and each new codec release would require reviewing the changes. Such modifications are better suited for being done in dedicated projects like Lancer or LAME MT.

The third method, however, could theoretically be implemented on a higher level – just run several instances of a codec in parallel and concatenate their output back together in the correct order.

This seemed reasonably easy and had the advantage that the same method might be usable for different codecs. I decided to try implementing this on top of the Opus codec first, because it has a constant block size and outputs raw packets that can be intercepted and reordered before being written to the output file.

How it works

For the first proof-of-concept implementation, I made the codec driver split the input data into blocks of several frames of audio data and distribute those blocks to worker threads in a round-robin manner. Except for very short files, there will usually be many more fragments than there are threads to be used, which allows for quicker distribution of work to the threads.

The worker threads then do their job in the background and collect packets of encoded audio data obtained from their respective codec instances.

When it's a threads turn for getting new data, the codec driver asks it for the collected audio data packets, writes those to the output stream and finally assigns a new block to the thread.

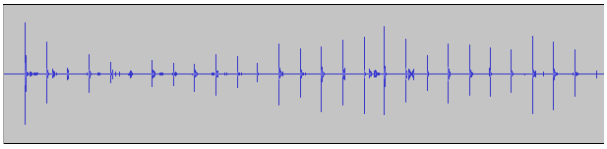
Basically, the inside of the main conversion loop looks like this:

```
Worker *worker = workers.GetNth(nextWorker++ % workers.Length());

/* See if the worker has some packets for us.
 */
if (worker->GetPackets().Length() != 0) ProcessPackets(worker->GetPackets(), ...);

/* Pass new frames to worker.
 */
worker->Encode(samplesBuffer, ...);
```

Details

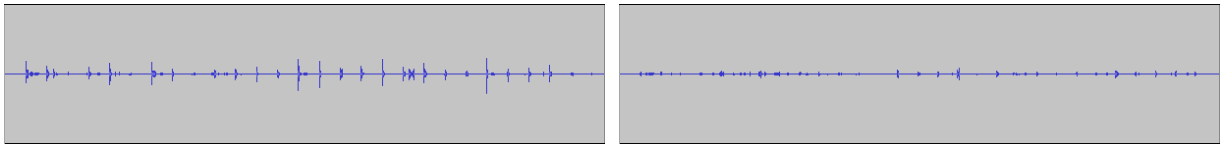
The naive implementation of splitting the audio data into several parts encoded by separate codec instances and concatenating the output back together, proved to be problematic, though.

As mentioned above, frames are not completely independent of each other. Transformations and filters applied during the encoding process need multiple frames worth of audio data to adapt, so concatenating packets created with different initial filter states can lead to heavy distortions.



Difference signal of single vs. multi-threaded conversion

To avoid this effect, the audio fragments to encode have to overlap by a few frames to give the filters enough time to adapt. After trying out different values, I settled with an overlap of 8 frames for an Opus frame size of 10ms or more (a larger overlap is necessary at shorter frame sizes).



Difference signals with 1 and 8 frames overlap

With an overlap of 4 or more frames there are no more distortions. The remaining differences are on the same level as when encoding the same file starting at different offsets and should be unnoticeable.

Resulting implementation

The current implementation of this method for Opus and AAC uses blocks of 128 frames with an overlap of 8 frames. That's a 6.25% or 1/16th overlap limiting the maximum possible speedup to 16x. In practice, though, even if you have a 16 core CPU, the actual speedup will be lower because of non-parallelized decoders and additional work needed for thread management.

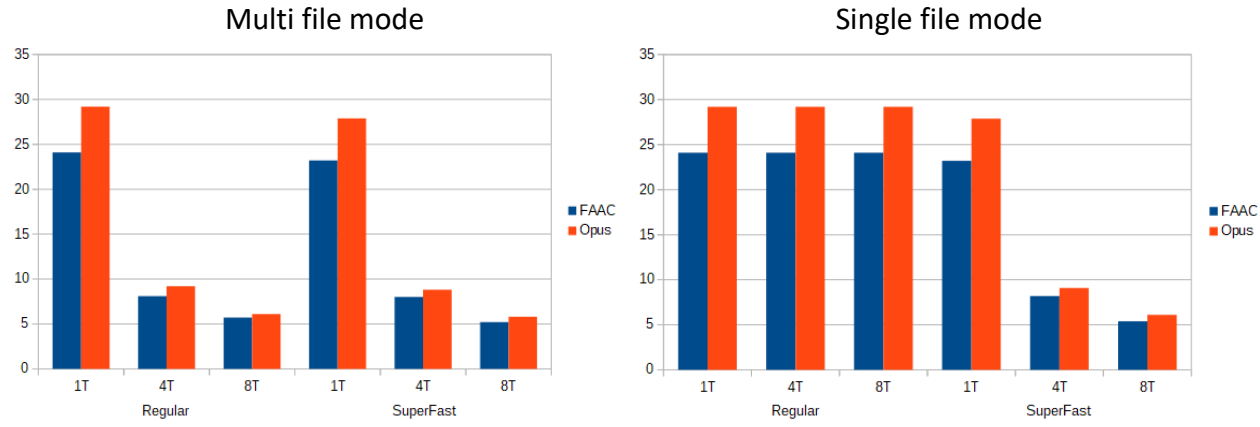
The multi-threaded codec driver solves the aforementioned problems of parallelizing conversions to a single output file and improving efficiency when converting less files than there are CPU cores.

Serial versus multi-threaded processing when converting to a single output file

Multi-threaded processing when converting less files than there are CPU cores

Performance numbers

The actually achieved speedups are quite significant. So good actually, that even on a 6 core CPU the conversion speed is often limited by decoding speed which makes me think about future possibilities of parallelizing the decoder side as well.



Speed comparison of regular vs. SuperFast codecs in multi and single file mode

In multi file mode, the SuperFast codec drivers are only slightly faster than the regular ones. Only one codec instance per track was used in this test and the performance advantage solely stems from decoding and encoding being done in separate threads.

In single file mode, though, the SuperFast codecs show their full potential. Using four threads, the conversion is already three times faster than with the regular drivers, which use only a single thread in this mode.

Note that the multi-threaded Opus encoder will exhaust its full potential only with 48 kHz audio data as that is the only sample rate natively used by Opus. The resampler used for feeding other sample rates to Opus still runs in non-parallel mode. I'm looking into ways of parallelizing it too.

What's next?

I chose Opus and AAC for the initial implementation of this idea, because they use constant frame sizes and straight forward frame packing. Thus, it was relatively easy to implement this idea on top of them without having to worry about too many side effects.

Adapting this method to other AAC codecs like FDK-AAC or other codecs with constant frame sizes like Speex should be relatively easy and will be done soon.

After that, the next step will be implementing this technology on top of the LAME MP3 encoder, which will be a bit more challenging. In MP3 files, frames can actually start before their frame header to make use of unused bits from previous frames. Therefore, you cannot simply concatenate the output of the encoder instances, but need to unpack the frames first and re-pack them when writing the output stream.

Some other codecs don't lend themselves very well to this kind of optimization. These include Ogg Vorbis (because of how block size switching changes the resulting frame lengths) as well as FLAC and WMA (because their APIs do not allow intercepting packets before writing the output stream). It currently does not seem feasible to implement this technology on top of those codecs.

Downloads

Download an experimental fre:ac build with multi-threaded Opus, FAAC* and Core Audio converters:

- Windows: x86-64, i686
- macOS: Universal Binary (x86-64, i686 and PPC)
- Linux: x86-64, i686
- FreeBSD: x86-64, i686

* The FAAC codec is provided as a fallback when the Core Audio codecs are not available.

Source code

The source code for the multi-threaded codec drivers can be obtained at:

https://github.com/enzo1982/superfast

About the author

I am Robert Kausch, founder of the fre:ac project and main author of the fre:ac audio converter.

Living in Hamburg, Germany, I am working as a software developer and project manager for Human Resources business software in my main job and on the fre:ac project in my spare time.

I started fre:ac as a very simple audio converter and CD ripper tool mainly for myself and a couple of friends. Today, it's one of the most popular audio converters with hundreds of thousands of users.

Building fre:ac has been a great opportunity to be part of the worldwide Open Source community, learn about new technologies and improve my software development skills.

Feel free to contact me at robert.kausch@freac.org for any questions regarding my work.