Musical Source Separation on the TI TMS320C6748 DSP processor

Project report
EE 750 - DSP Hardware and Implementation

Submitted by

Mohammed Niyas
M A Rohit

November 2019

Table of Contents

Abstract

We present a musical source separation system implemented on a TI TMS320C6748 DSP floating point processor, that takes a mixture audio input and separates it into vocals and background music. At the heart of the system is a convolutional neural network that is trained beforehand on a computer to perform this separation. The learned weights are then loaded onto hardware and the operations involved in making a forward pass through the model when given a new song as an input are implemented. Design choices that influence the computation time and memory requirements are explored, the challenges faced in implementing the model in C are discussed, and the results obtained on hardware are compared with simulation results from MATLAB.

1. Introduction

Musical source separation(MSS) is the task of extracting the audio of any single source from an audio signal that contains different combinations of several sources playing simultaneously. For example, extracting the vocals from a stereo recording of a song that contains drums, guitar, keyboard, etc. in the background. If the mixture contains as many channels as the number of sources and the mixing process is known and fixed, then the task may not be that difficult, however this is not the case in most practical scenarios, and MSS then involves understanding the unique properties and the structure of the individual sources and using this knowledge to break the mixture down into its constituent parts.

1.1 A general MSS system

Source separation methods have traditionally focused on using the time-frequency representation known as the Short-Time Fourier Transform (STFT), instead of the time-domain audio signal. This allowed the use of effective matrix decomposition techniques like Non-negative Matrix Factorization (NMF), which when used on the magnitude STFT, also known as the magnitude spectrogram, of the mixture, gave as outputs the spectral slices of the different sources and their corresponding activations which indicated when each of the sources was active. These would then be used to compute time-frequency masks for each source, and applied to the input mixture spectrogram as filters to obtain a separate magnitude spectrogram output corresponding to each source. These output spectrograms could then be combined with a phase spectrogram - either from the input mixture itself or estimated iteratively, to produce complex STFTs, which when inverted would give the real time-domain audio signals for each source.

The block diagram in figure 1 shows the components of a general MSS system as described above. A brief explanation of each of the above components follows

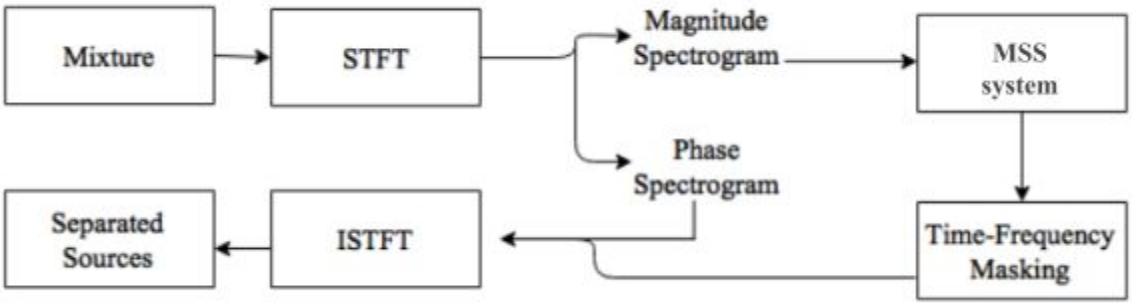

Figure 1: Block diagram of a general MSS system (Chandna, Pritish, et al. 2017)

1.1.1 STFT

The discrete STFT is a time-frequency representation obtained by the repeated use of a Discrete Fourier Transform(DFT) on smaller portions of the audio. This is useful when the audio signal is not stationary and has a time-varying frequency spectrum. In such a scenario, it is not meaningful to take the DFT of the entire signal, rather to break it down into smaller portions over which it can be assumed to be stationary. In practice, these smaller portions usually overlap with each other by a fixed amount and are also multiplied by a tapered window. Figure 2 illustrates this.

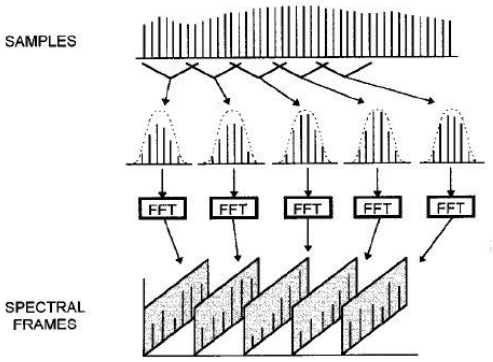

Figure 2: The STFT

Figure 3 shows what a typical magnitude spectrogram looks like. The horizontal axis is time and the vertical is frequency, while the intensity indicates magnitude. The spectrogram basically shows the evolution of different frequencies across time.

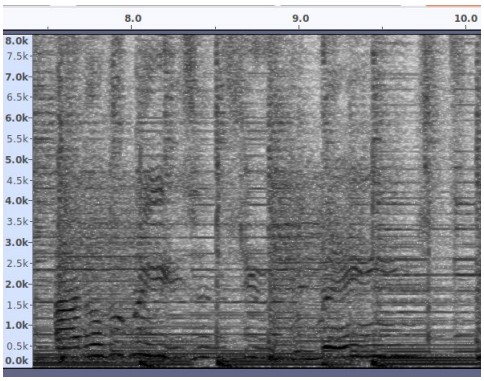

Figure 3: An example of a magnitude spectrogram

The operation of STFT can be shown mathematically through the following equations. First, the windowed signal $s(m,n)$ is obtained by windowing a signal $x(n)$ with the window $w(n)$. Then the STFT $S(m,w)$ is obtained by taking the DTFT of $s(m,n)$.

$$s(m,\ n)\ =\ x(n)\cdot w(n-m\cdot N/2)$$
$$S(m,\ \omega)\ =\ DTFT\{s(m,\ n)\}$$

1.1.2 ISTFT

The inverse STFT is the inverse process of the STFT and is used to obtain the corresponding time-domain signal. In the case of our task, since we are working with real signals in the time domain, the STFT is necessarily complex valued and conjugate symmetric. Therefore, after the MSS system gives as the magnitude spectrogram outputs, the phase components need to be added and the complex STFT needs to be made conjugate symmetric before inverting. This is to ensure that the ISTFT results in a real time-domain signal. Therefore in the inverse we have the following equations:

$$s(m,\ n)\ =\ DTFT^{-1}\{S(m,\ \omega)\}$$

$$\widehat{x}(n)\ =\ \sum_{m}\ [\ s(m,\ n)\cdot w(n-m\cdot N/2)\ ]$$

Where $\widehat{x}(n)$ is the reconstruction of $x(n)$.

1.1.3 The MSS system

Traditionally, NMF-based methods have dominated the source separation realm for a long time and with a good amount of success under certain conditions. However, with the advancement of deep learning, and their effectiveness in a wide variety of tasks, state of the art results have been obtained using different kinds of deep neural network models. Furthermore, while NMF-based methods need to iteratively solve for the sources each time for a given input, neural network models can first be trained on a training dataset to learn the weights, after which at test-time, a forward pass through the model is all that is required to produce the outputs for any given input. This makes them good candidates for low-latency applications. However, all is not that well yet. To obtain satisfactory outputs, the neural network models need to be deep and complex enough, which often results in a lot of operations even during test-time. And while this is not a problem if the model is run on a modern-day pc with GPUs that enable superior parallel processing, a piece of hardware such as a DSP board poses severe constraints.

1.1.4 Time-frequency masking

Masking is a step that is introduced to ensure that the separate source spectrograms estimated by the MSS system add to the original. This is ensured by first deriving a mask for each source from the estimated spectrograms and then applying this mask on the input mixture spectrogram, rather than using the estimated spectrograms directly.

2 Proposed system

For our implementation, we use the MSS system introduced by Chandna, Pritish, et al. (2017). This is a deep convolutional network model designed to have one encoder and decoder branches, one for each of the two sources to be estimated - vocals and background. The encoder consists of two convolutional layers with a pooling layer in between, followed by a fully connected layer. The decoder reconstructs two separate matrices of the same dimensions as the input, by performing a sort of an inverse of the operations performed by the encoder. Figure 4 shows the architecture of the CNN model.

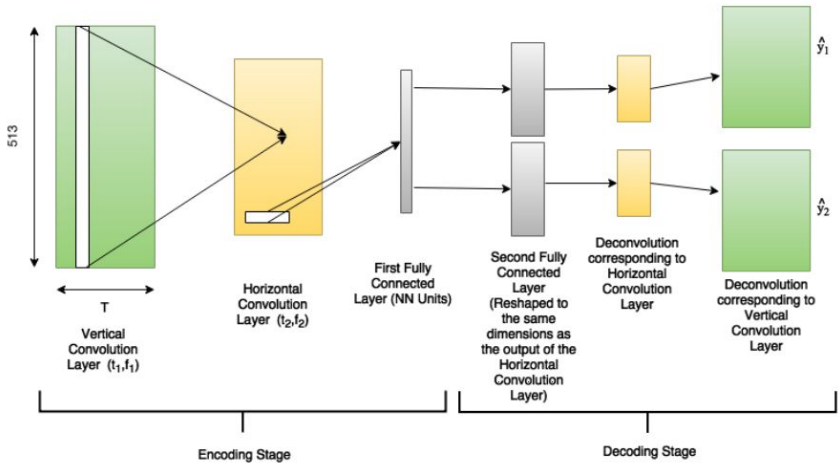

Figure 4: The CNN model architecture (Chandna, Pritish, et al. 2017)

2.1 Input and output

The input to the network is a magnitude spectrogram chunk containing a few frames of the audio - ie., the spectrogram of the entire audio signal, which is of dimensions NxM, is broken down into overlapping chunks containing 30 DFT frames each. Therefore, the input to the network is of size Nx30. Here, N is related to the number of points in the DFT, which will be specified in the section below.
The output is two matrices of dimensions Nx30, each representing a different source.

2.2 Initial model parameters and training

The model architecture as implemented in the paper is as follows. The first layer is a vertical convolutional layer containing 30 filters of dimensions 1x30, (1 in time, 30 in frequency), with a stride of 3. This is followed by a max pooling of dimensions 1x4, and then a second convolutional layer containing 30 filters of dimensions 10x20, with a stride of 1. Finally, a fully connected layer of 256 units completes the encoder portion of the network. The decoder contains 2 sets of mirrored copies of the above 4 layers. While the convolutional layers in the encoder and decoder are tied, ie., they have the same set of weights, the 3 fully connected layers all have different learnable weights. Training is performed by giving spectrogram chunks of the mixture audio as input, and the corresponding chunks of the clean sources as the target.

2.3 Other algorithm parameters

STFT

- Hanning window of length 1024 samples, hop of 512 samples
- At fs =16kHz, this amounts to nearly 64 ms and 32 ms
- DFT size of 1024

CNN

- Spectrogram chunks of 30 frames(0.96s), with an overlap of 10 frames(0.48s) between successive chunks.

3. Preliminary MATLAB Simulation Results

The authors of Chandna, Pritish, et al. (2017) have made available a model trained on the MusDB dataset, which is used for our initial simulation experiments on MATLAB. While the provided code is in python and makes use of deep learning libraries and GPUs for speedup, the code had to be re-written in a manner that could be implemented on hardware. Hence, we first set out to code the entire model on MATLAB, without the use of any libraries and in a manner that resembles C - ie., without utilizing any vectorization of variables or parallel processing of operations, or the use of in-built functions in MATLAB.
By “coding the entire model”, we mean, implementing each of the layers of the model in separate blocks to perform the same operations. Since a pre-trained was used model for the initial experiments, all the required weights were available, and we had to only write the set of operations to be performed to process

a given audio signal to produce two audio signals corresponding to each of the sources - vocals and background.

We were able to obtain the same results as claimed in the paper and as obtained from the shared python codes in terms of the separation quality, but the code took several times longer to run. Our initial experiments on a 15s audio clip took nearly 30 minutes to complete. The profiler on MATLAB was used to identify the most time-consuming part and it turned out to be the convolution operation. This was to be expected, as out of the total of 9 layers in the model (encoder + decoder), 6 are convolutional layers.

4. Hardware implementation

4.1 Considerations for hardware implementation

The duration it took for the implementation on MATLAB made the algorithm unsuitable for hardware. Hence the following changes were made:

1. Simplifying the architecture
   The task of simplifying a deep convolutional network while keeping the performance unaltered is not a trivial task, and generally requires clever manipulations and several experiments. In this work, however, only a few straight-forward tweaks to the model are made while repeatedly re-training the model with the updated parameters and qualitatively verifying the separation performance. The changes made are as follows:
   a. The number of filters in each of the convolutional layers reduced to 10 from 30.
   b. The 2-dimensional kernel in the second convolutional layer changed from 10x20 to a one-dimensional kernel of size 1x20.
   c. Number of hidden units in the first fully connected layer reduced to 64 from 256.

2. The training dataset contained audio files sampled at 44.1KHz. We downsampled all these audio files to a rate of 16KHz, to further make the implementation faster. Re-training the model with audio files of 16kHz to make the model work in test-time on 16kHz was necessary because the same time dimensions in each case do not carry the same amount of audio in seconds. In other words, a 1024 point window would contain different durations in seconds of the audio signal.

The complexity of the model involved led us to the following decisions about the hardware and hardware settings to use:

1. TMS320c6748 DSP was chosen for its capability of performing floating-point operations. While fixed point operations are faster and the audio read can be represented equally well in fixed-point format, the model weights can often be very small values less than 1, and implementing the whole system in fixed point would require more careful verification at each step. This was not explored in this work, and all variables were declared as float type.

2. The board was operated at its maximum clock rate of 456MHz. This was achieved by modifying the appropriate block in the GEL file, as outlined in [2]. The GEL file can be found at

<C:\ti\ccsv5\ccs_base\emulation\boards\lcdkc6748\gel\C6748_LCDK.gel>.

3. There were also two other choices of hardware available - c5515 fixed-point DSP and c6678 multi-core DSP. The c5515 board was not preferred because of its fixed-point nature and because of the lack of sufficient RAM. The model weights in our task, after simplification of the architecture occupied a total memory of about 5 MB, while the c5515 only has about 4MB of RAM.
   As for the c6678, the board has no ports for audio input and output. This prevents the possibility of directly interacting with the board both to give the input from an audio device as well as to listen to the separated audios through earphones or speakers.

4. To perform FFT and IFFT, TI's DSPLIB library was made use of as it provides optimized codes for these operations. The library also comes with a number of other operations commonly found in DSP applications, such as convolution, vector addition, dot product, etc. While the convolution can be easily used, most of the other operations require the inputs to be of ***constant*** type, making the library their usage a little restrictive.

4.2 Issues faced and solutions found

1. Due to the lack of sufficient memory in the SH- and D- RAMs, all the model weights were loaded onto the DDR memory. The weights were loaded during compilation, from a header file. However, due to the code residing in the internal memory, the compiler kept throwing errors about not being able to find the required variables stored in the DDR memory. Hence, all the components(code, stack, variables, etc) in the linker cmd file were moved to the DDR RAM.

2. The CNN architecture was originally developed for image-related applications, like digit recognition, image classification, etc, and hence TI has provided an IMGLIB library with optimized codes specially for such applications. These are useful if the data is an image with 8-bit (fixed point) data, because then, the processor can utilise its multiplier units which are capable of performing multiple fixed point multiplications in parallel to speed the process up. However, our application makes use of float data, so we did not expect to achieve appreciable speed up with the use of this library, hence its use has not been explored in this work.

3. While compiling the code, there were errors with using the default math library functions provided in Code Composer Studio's version 6. Operations like sin(), cos(), arctan(), sqrt() etc returned erroneously large values. Hence, a separate library provided by TI, called MATHLIB was installed and used.

4. Due to the large number of variables involved in the processing of the audio in the main CNN part of our code, declaring all these large variables inside the main function often led to memory read issues, where the code execution would go into invalid locations of the instruction memory. Hence, all the variables had to be declared globally, at the top of the code, below all the include

operations.

5. In order to avoid the use of too many loops for the different blocks of the CNN, some steps were implemented together, For instance, reshaping operations are quite common in a CNN which contains fully connected layers. Hence, to avoid calculating an output variable first in its normal output size and then resizing it in another loop, the variable was initialised with the shape that would result after resizing, and values were stored by indexing the appropriate locations.

Final algorithm implemented

- Load trained weights and acquire full-length mixture audio at 16kHz
- Loop over the audio signal
    - STFT: obtain overlapping frames of 60ms each -- multiply with suitable window function -- compute 1024-point DFT - accumulate 30 frames into one chunk
    - Input chunk to CNN, obtain two chunks out
    - Compute masks and filter the chunks of each source
    - Save each chunk in an overlap-add manner
- Inverse STFT: Multiply with phase - make conjugate symmetric - obtain frame-wise IDFT - overlap-add with suitable window to obtain separate audio signals

5. Results and Discussion

After the modifications made to simplify the architecture, the MATLAB simulation was able to run in real time on a PC with a superior processor and a high clock rate of about 2GHz. This did not reflect on the hardware however, and it ended up taking nearly 15 minutes to process a 10 second clip of input audio. Hence, instead of trying to acquire the audio and process it simultaneously, we implemented a three-step process:

1. Acquire full-length audio
2. Process
3. Playback

While the ideal goal of performing real-time MSS was not achieved, the implementation itself was successfully performed, and the outputs obtained at every step of the process on the hardware matched the simulation results exactly up to 3 decimal places of the values. A detailed timing information of each module in the MSS portion is given below. These are the times taken per chunk of the audio.

1. Computing STFT - 2s
2. Encoder
    a. 1st convolution layer - 2.8s
    b. Pooling - 1s
    c. 2nd convolution layer - 2.7s
    d. Fully connected layer - 1.1s

3. Decoder
    a. 2 fully connected layers - 1.8s
    b. 2 deconvolutional layers - 9.7s
    c. 2 inverse pooling layers - 1s
    d. 2 deconvolutional layers - 15s

This shows that the total time taken per chunk(about 1s of the audio at 16KHz sampling) is about 35s, of which most of the time is consumed in the last two deconvolutional layers. One solution for this, since all the layers after the model simplification only have 1D convolution operations, is to make use of the optimized implementations of convolutions provided in the DSPLIB library. That would be a good starting point for improving the computation time taken by the system.

6. References

[1] Chandna, Pritish, et al. "Monoaural audio source separation using deep convolutional neural networks." *International conference on latent variable analysis and signal separation*. Springer, Cham, 2017.

[2] TMS320 C6748 Increase Clock Speed, URL: http://e2e.ti.com/support/processors/f/791/t/419316?TMS320-C6748-Increase-Clock-Speed