Comparative Study on FIR Filter Coefficient Compression

AYUSH BHAT LEONARDO MANCA
ayushb | lmanca @kth.se

January 16, 2023

Abstract

Finite Impulse Response filters are widely used in signal processing applications. Most of these applications are deployed on embedded systems which are resource constrained and thus require efficient use of memory to store the filter's coefficients. Although some patented algorithms involving storage of successive-differences have been implemented to perform the coefficient's compression, a comparison between these methods and popular compression techniques such as FLAC have not been investigated yet. This research analyses the performance of these compression methods applied in different applications and their relative results in each field. It was found that The successive-difference method gives compression ratios from -40% up to 80%, varying greatly with the FIR filter type. This variance is attributed to the algorithm not considering coefficient signs, and a method to resolve this is presented in this project. Further, it was found that compression algorithms such as FLAC are able to consistently perform well for FIR filter coefficient compression, providing over 50% compression ratio for a wide range of FIR filter types, however the decoders are more complex to implement in hardware, compared to the successive-difference method.

Contents

List of Acronyms and Abbreviations

FIR: Finite Impulse Response
LPF: Low Pass Filter
HPF: High Pass Filter
BPF: Band Pass Filter
FLAC: Free Lossless Audio Codec
ALAC: Apple Lossless Audio Codec
JPEG: Joint Photographic Experts Group
FLIF: Free Lossless Image Format
HEIF: High Efficiency Image File Format
LPC: Linear Predictive Coding

1 Introduction

In the field of digital signal processing, Finite Impulse Response (FIR) filters are widely used for applications ranging from speech processing, channel selection and separation, to bio-electric signal processing [1]. For sharp transition FIR filters, a large number of filter coefficients need to be stored in memory [2]. This poses a problem for resource constrained embedded systems, which may only have a limited amount of memory and could thus benefit from an efficient way of storing these coefficients. Compressing an FIR filter's coefficients before storage would allow the system to have a higher number of digital filter instances as well as free up memory for other tasks. The current state-of-the-art implementation for FIR filter coefficient compression involves the storage of successive differences between adjacent coefficients [3]. This method allows for relatively simple hardware based encoding/decoding of the coefficients but higher levels of compression require recursive storing of the differences between coefficient differences and so on. For being optimal, compression techniques must exploit certain characteristics of the data being compressed. For example, file formats such as the Free Lossless Audio Codec (FLAC), which are used for lossless audio file compression, exploit the sample to sample correlation inherently present in audio files [4]. This sample to sample correlation is also seen between FIR coefficients and thus the techniques used in FLAC could prove to be optimal for compressing FIR coefficients.

This research will focus on lossless compression algorithms applied to FIR filter coefficients. Each compression algorithm is optimised depending on the nature of the data to be compressed. The most common use cases for data compression can be found in audio and image processing, and some popular compression techniques have been developed and are widely used today. FLAC is an audio coding format for lossless data compression, specifically designed for storage of audio files [4], where the compression ratio ranges between 36.2% to 68.4% of the original uncompressed data [5]. Apple Lossless Audio Codec (ALAC) is an audio coding format for lossless data compression which offers similar characteristics compared to FLAC such as a compression rate that clusters around 50% [6]. Furthermore, benchmarks compared to FLAC show that the CPU usage is about 4 times higher [7]. Monkey's Audio is an algorithm for lossless audio data compression. Compared to formats such as FLAC it achieves better compression performances, offering approximately a 41% reduction of the original uncompressed data [5]. Image compression algorithms include formats such as Joint Photographic Experts Group (JPEG), Free Lossless Image Format (FLIF) and High Efficiency Image File Format (HEIF). However, since they are suited for 2-dimensional data compression, they are not considered in this project.

1.1 Theoretical Framework

FLAC exploits sample to sample correlation in audio files, for compression. Since FIR filter coefficients also exhibit this correlation, we decided to compare the successive-difference method to FLAC. For comparing these two compression techniques, an in-depth study was done to understand the mechanism of each technique.

1.1.1 Successive-Difference Compression

Compression of FIR filter coefficients based on the successive difference method relies on the fact that the difference between two adjacent coefficients is much smaller than the values of the coefficients themselves. In this method, the first sample in a series of FIR coefficients is stored as-is in memory, followed by storing the difference of the first and second sample, second and third sample, and so on. The coefficients are recovered by adding the differences to a previous sample. This can be applied recursively to compress the coefficients further. In this text, we refer to a single successive-difference as First Order, a successive-difference of the first order compression as Second Order and so on. However storing higher-order differences would require more decoding hardware and thus storing higher than second or third-order differences does not drastically improve the compression with respect to the chip-area required for implementing the decoder.

1.1.2 FLAC Compression

FLAC is an extension of the SHORTEN algorithm [8], which uses a set of predictors on a block of samples at a time (Fig. 1). An appropriate predictor is chosen to predict the samples in a block to the nearest mathematical model, and the residual is coded through Rice encoding. The first stage of the algorithm divides the sample stream into blocks/frames of fixed sizes and needs to be of an appropriate length. If it is too short, the frame headers (which contain parameters for the linear predictor) would cause too much overhead, and if it is too long, an appropriate mathematical model describing the samples may not be found. The second stage is the linear predictor, which decorrelates the samples within a frame. By the second stage, the samples in a frame are losslessly encoded as the linear predictor parameters, and the residual or error is sent to the third stage for entropy coding. In typical lossless audio compression, the entropy coder utilizes Huffman, run-length or Rice coding [9].

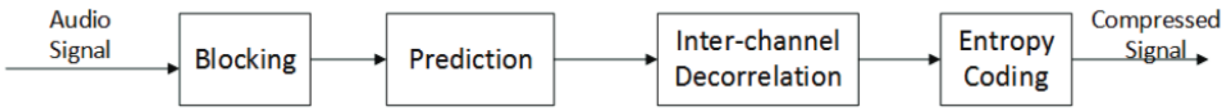

Figure 1: Structure of FLAC encoder [10]

1.2 Research questions and hypothesis

This project aims at answering the following questions:

1. Does FLAC compression perform better than successive-difference compression?
2. What are the limitations in the current state-of-the-art algorithm and can it be improved?

We hypothesize that the compression techniques used in FLAC would be able to exploit the inter-sample correlation and result in a better compression ratio, with respect to the current state-of-the-art compression technique for FIR filter coefficients.

2 Method

This section presents the procedures, sources of test data, tools used to process the data, as well as the experiments performed to compare the performance between the successive-difference algorithm and FLAC.

2.1 Procedures

First, we generated sets of suitable test data, that covered a wide range of filter coefficients used in real-life scenarios. This includes coefficients for different filter types such as low-pass filters (LPF), high-pass filters (HPF) and band-pass filters (BPF). For each type of filter, we also generated a dataset of a small number of coefficients, and an intermediate number of coefficients.

Furthermore, we implemented the encoders for the successive-difference algorithm and the FLAC algorithm, followed by a verification of their functional correctness. Once this was done, the test data was fed to the encoders and the compression ratios for each case were computed. A comparison of the number of uncompressed and compressed bits was then represented graphically.

2.2 Tools

For generating filter coefficients, we used TFilter, which is a free to use filter design software available online. The tool enables us to set stop/pass-bands with different transition bands, allowing us to generate a wide range of input scenarios for the compression algorithm.

The compression algorithms were implemented using Python, extended with the Numpy and Matplotlib libraries for processing and presenting the data. We also used the pyFLAC library, which is a Python wrapper around the popular libFLAC library, for implementing the FLAC encoder.

2.3 Experiments

This subsection is an overview of the methods used for generating test data and the implementation method for the encoder and decoder for the successive-difference and FLAC algorithms.

2.3.1 Test data generation

To cover a wide set of test inputs, 6 different FIR filters listed in Tables 1 and 2, have been tested. Additionally, an impulse response modelling the acoustic response of a physical space has also been tested to demonstrate the performance of the algorithms when the number of coefficients is very large.
The filters have been designed with a sampling frequency of 44100 Hz, and the gain for each pass-band has been set to 1, while for each stop-band has been set to 0. Furthermore, the stop-band ripple for the filters 1 and 2 has been set to -20dB and -40dB respectively, while the pass-band ripple has been set to 1dB for both.

Table 1: Low pass and High pass filter configuration (1dB pass-band ripple)

FIR Filter	LPF 1	LPF 2	HPF 1	HPF 2
Pass-band[kHz]	0-0.4	0-5	0-15	0-17.9
Stop-band[kHz]	0.5-22.05	6-22.05	16-22.05	18-22.05
Taps	43	681	43	669

Table 2: Band pass filter configuration (1dB pass-band ripple)

FIR Filter	BPF 1	BPF 2
Pass-band [kHz]	6-7	6-7
Stop-band 1[kHz]	0-5	6-22.05
Stop-band 2[kHz]	8-22.05	7.1-22.05
Taps	51	699

2.3.2 Algorithm implementation

Two Python scripts were written in order to implement an encoder and a decoder for the successive difference algorithm and FLAC using the numpy and the pyFlac libraries respectively (see Appendix A.1 and A.2). The test data was fed to the encoders and the compression ratios for each case were computed. A comparison of the number of uncompressed and compressed bits was then represented graphically. For computing the compression percentage the formula used is:

$$Compression\% = (1 - B_c/B_u) * 100 \tag{1}$$

Where, B_c is the number of compressed bits and B_u is the number of uncompressed bits.

2.3.3 Optimisation

It is unclear whether the successive-difference algorithm used in the patent is only for unsigned coefficients. In fact, the algorithm seems to neglect the sign of the coefficients and thus, requires some modifications in order to be used with signed coefficients, which are typically used for FIR filters.

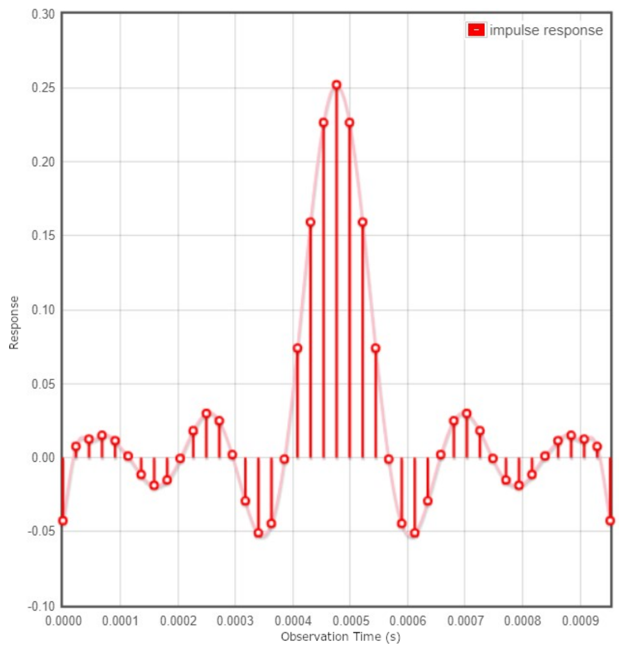

Figure 2: Impulse response of the LPF 1

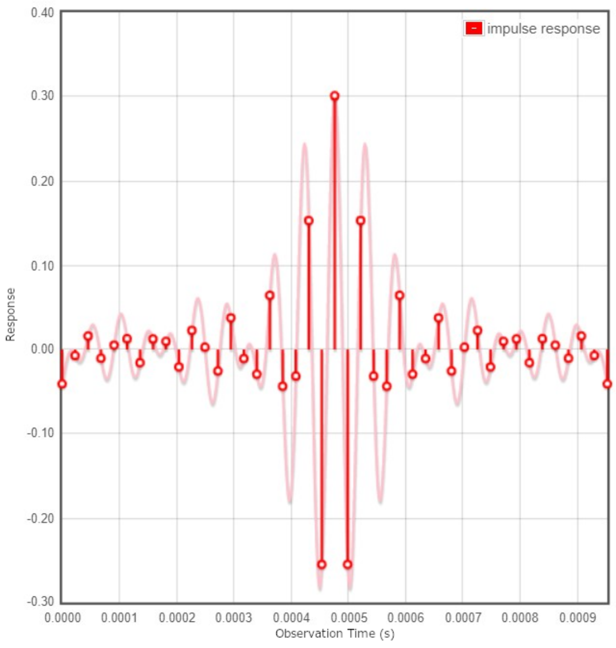

Figure 3: Impulse response of the HPF 1

As shown in Fig. 2, the majority of the adjacent coefficients (dots in the graph) have the same sign, while Fig. 3 showcases how the majority of the adjacent coefficients have different sign. While for LPF 1 using a successive difference would yield smaller numbers, it would result in the opposite for the HPF 1. Reasoning on these considerations, we decided to provide a simple modification in order to optimise the algorithm: using an addition operation whenever two adjacent coefficients have a different sign. This way the result of the computation would certainly lead to a smaller number. For example we could consider the following coefficients:

$$C = \{10, -8, 1000, -900, 33\}$$

Performing an overall subtraction would result in bigger numbers, for example:

$$1000 - (-900) = 1900$$

requires more bits for storage than

$$1000 + (-900) = 100$$

In order to retrieve the original coefficients, an array of boolean values keeping track of every operation for a pair of adjacent coefficients was used. Even though this approach was very promising, it led to worse results. The cause behind this problem has to be found in the size of the boolean array, which is equal, in bits, to the number of taps in the filter minus 1. In the end, this approach causes a higher memory overhead than the simpler, non-optimized, approach.

However, as Fig. 3 highlights how most of the adjacent coefficients might have a different sign, thus, performing a global successive addition could yield better results in some test cases. Thus, the algorithm has been improved by adding a predictor that performs an addition instead of a subtraction in case there, is a majority of adjacent coefficients with opposite signs. Results show how this method overall provides better results, especially for high pass filters, and neutral results for Band Pass Filters as well as the long impulse response.

3 Results and analysis

In this section the two compression methods (FLAC and successive-difference) are benchmarked. In the successive difference method, the compression is measured in terms of bits: the uncompressed coefficients size is computed in bits and later compared to the size, also in bits, of the compressed coefficients. This is done up to the third order of successive-differences. On the other hand, FLAC's performances are measured in terms of bytes, since the pyFlac library returns compressed data in terms of byte frames.

3.1 Successive difference compression

As mentioned in subchapter 2.3.1, 6 different filters and an impulse response have been used as input to the algorithm. The following results show how the algorithm performs impressively well (as shown with Fig. 5) sometimes, while other times it performs very badly (as shown in Fig. 7), resulting in bigger coefficients than the original, uncompressed ones (Fig. 6 to Fig. 8).

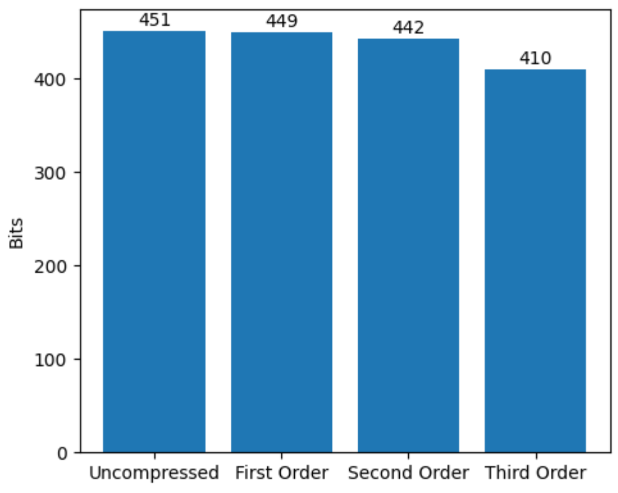

Figure 4: Comparison of uncompressed and compressed bits for LPF 1

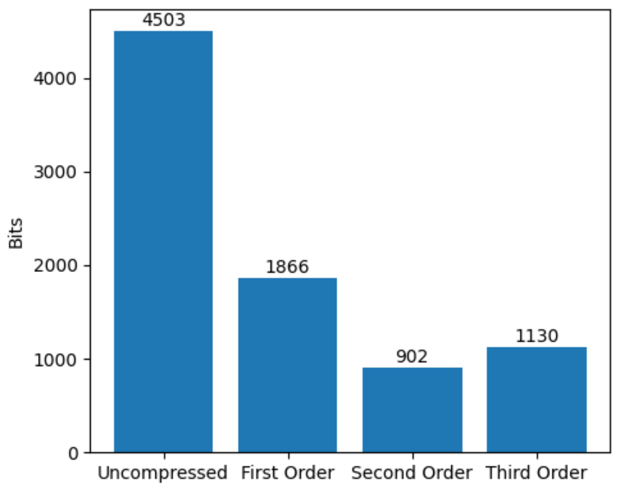

Figure 5: Comparison of uncompressed and compressed bits for LPF 2

Fig. 4 showcases the performances in terms of bits for the LPF 1. The compression ratio of the filter depending on the order chosen has been reported in Table 3. The results show poor performances, that slightly improve by increasing the filter order. The poor results can be attributed to the few coefficients used by the filter but can most importantly be amputated to the algorithm not taking into account the sign of adjacent coefficients (sign neglection). On the other hand, Fig. 5 shows how the algorithm performs extremely well with LPF 2, as shown by Table 3. The reason behind the high performance of the algorithm with the LPF 2 can be found in the adjacent coefficients being mostly of the same sign as well as the big number of coefficients used.

Table 3: Compression ratio for LPF 1 and LPF 2

Compression Ratio	Raw	1° Order	2° Order	3° Order
LPF 1	0%	0.44%	2.00%	9.09%
LPF 2	0%	58.56%	79.97%	74.91%

Fig. 6 shows how the sign neglection plays a significant role in the performance of algorithm, as shown by Table 4, in which the compressed data actually requires more space compared to the raw one.

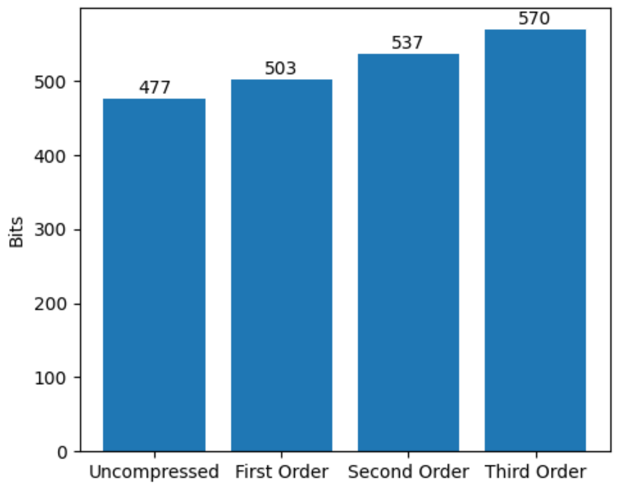

Figure 6: Comparison of uncompressed and compressed bits for HPF 1

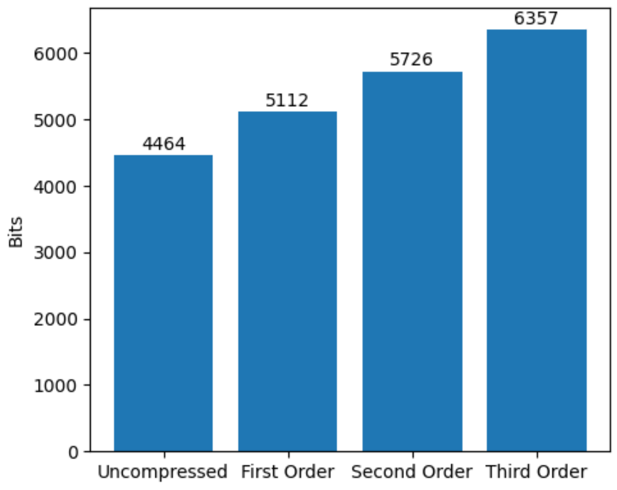

Figure 7: Comparison of uncompressed and compressed bits for HPF 2

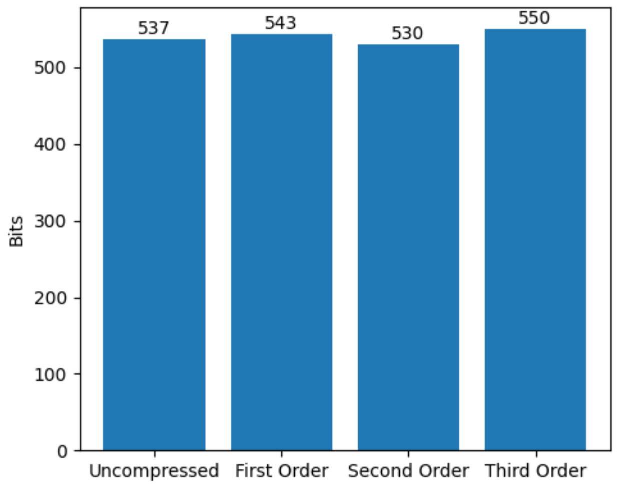

Figure 8: Comparison of uncompressed and compressed bits for BPF 1

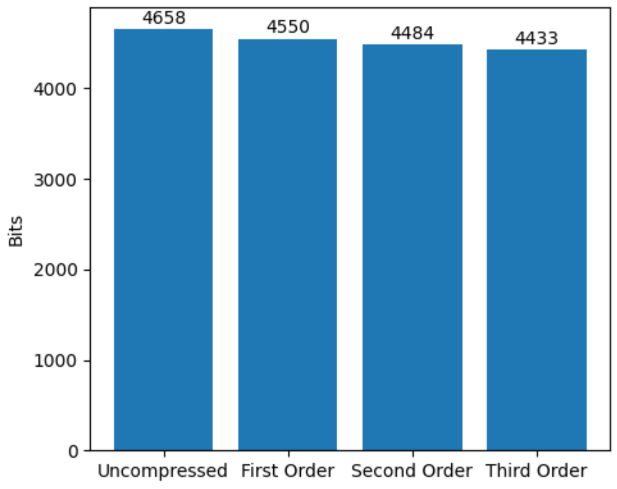

Figure 9: Comparison of uncompressed and compressed bits for BPF 2

Table 4: Compression ratio for the HPF 1

Compression Ratio	Raw	1° Order	2° Order	3° Order
HPF 1	0%	-5.45%	-12.58%	-19.50%
HPF 2	0%	-14.52%	-28.27%	-42.41%

The algorithm's sign neglection's impact on performances is especially highlighted by Fig. 7. As Table 4 shows, the performances drop dramatically, resulting in a array of coefficients requiring 42.41% more space than the original uncompressed data. On the other hand, BPF 1 (Fig. 8), shows performances that cluster around -2% to 1% as shown by Table 5. Thus the performance of the algorithm can be considered very neutral for band pass filters. However, this behaviour still highlights the bad efficiency of the algorithm.

Table 5: Compression ratio for the BPF 1 and BPF 2

Compression Ratio	Raw	1° Order	2° Order	3° Order
BPF 1	0%	-1.12%	1.30%	-2.42%
BPF 2	0%	2.32%	3.74%	4.83%

BPF 2 (Fig. 9), shows a slight improvement in terms of performance compared to BPF 1 (Fig. 8). This behaviour can be attributed to the high number of coefficients in the filter, and by a majority of adjacent coefficients having the same sign.

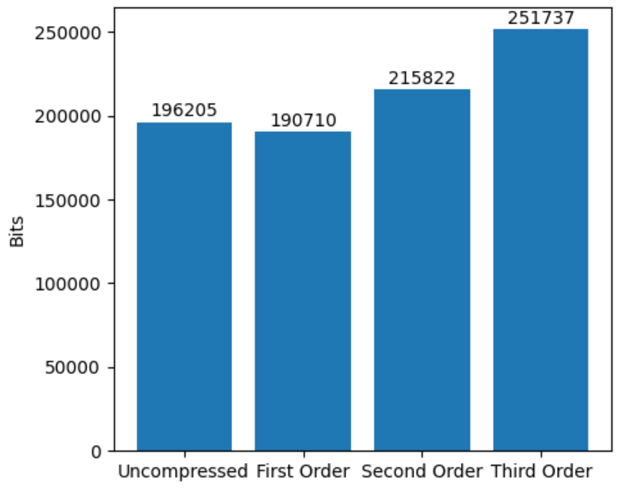

Figure 10: Comparison of uncompressed and compressed bits for an impulse response

The last experiment has been done on a long impulse response(IR) (Fig. 10). Table 6 shows how the algorithm provides bad performances that decrease as we increase the successive-difference order. This behaviour can be traced once again to sign neglection.

Table 6: Compression ratio for an impulse response

Compression Ratio	Raw	1° Order	2° Order	3° Order
IR	0%	2.80%	-10.00%	-28.30%

As mentioned throughout the subchapter, the cases in which the algorithm shows poor performances, mostly depend on the algorithm not taking into account the sign of the adjacent coefficients. In fact, as mentioned in subchapter 2.3.3, the implementation of the algorithm according to the paper [3], ignores the possibility of two adjacent coefficient having a different sign. This characteristic of the algorithm has a small impact with few taps, however, it has a much bigger impact by increasing the number of coefficients used by a filter. The latter explains the very poor performances of the HPF 2, and the need of an optimised version of the algorithm.

3.2 Optimised successive difference compression

As mentioned in subchapter 2.3.3, the optimised version of the successive difference algorithm performs a successive difference or addition by scanning the array of uncompressed coefficients: if the majority of the adjacent coefficients have different signs it performs a global successive addition, otherwise a global successive difference. Table 7 shows a performance comparison of the original algorithm and the optimised one for each filter.

Table 7: Comparison of compression ratios between original and optimised algorithm

Algorithm	Original			Optimised		
Compression Ratio	1° Order	2° Order	3° Order	1° Order	2° Order	3° Order
LPF 1	0.44%	2.00%	9.09%	0.22%	1.55%	8.43%
LPF 2	58.56%	79.97%	74.91%	58.538%	79.92%	79.17%
HPF 1	-5.45%	-12.58%	-19.50%	2.94%	5.45%	3.35%
HPF 2	-14.52%	-28.27%	-42.41%	11.78%	23.99%	34.99%
BPF 1	-1.12%	1.30%	-2.42%	-1.30%	0.93%	-2.98%
BPF 2	2.32%	3.74%	4.83%	2.30%	3.69%	4.77%
IR	2.80%	-10.00%	-28.30%	-5.79%	8.68%	1.42%

The table shows performance improvement which can be especially seen for HPF 2. The reason behind the slight decrease in performance for the LPF 1 can be found in the extra bit used in order to choose whether to perform a successive difference or a successive addition. Overall, the optimised version of the algorithm gives in general better results, and could thus replace the original one.

3.3 FLAC compression

It was seen that for a low number of coefficients, the FLAC metadata overhead exceeded the compression benefits and as a result, the input coefficient data of 86 bytes (43 coefficients) required 158 bytes of storage (Fig. 11). For a higher number of coefficients however, the FLAC encoding scheme works very well, with 1362 bytes of coefficient data (681 coefficients) being compressed to just 336 bytes, with a compression ratio of 75.33% (Fig. 12).

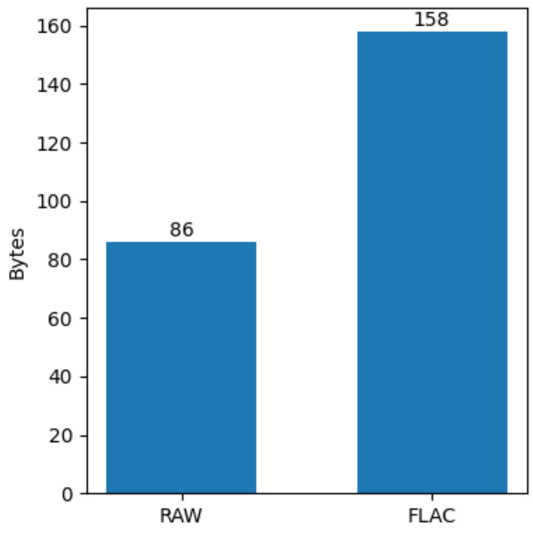

Figure 11: FLAC compression for LPF 1

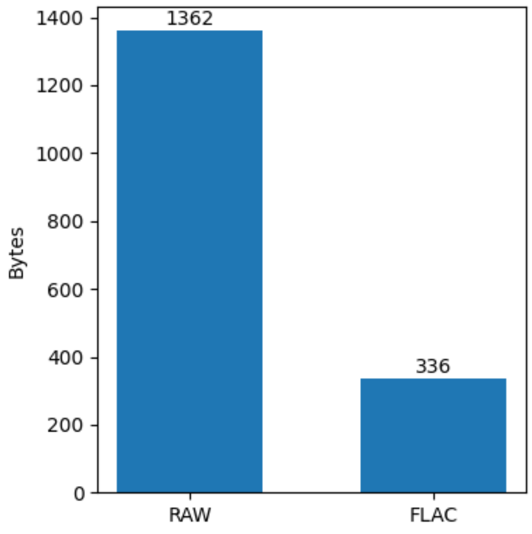

Figure 12: FLAC compression for LPF 2

Similarly, for the other filter types with a higher number of coefficients, FLAC performs consistently well, with compression ratios of over 50% (Fig. 13 to 15).

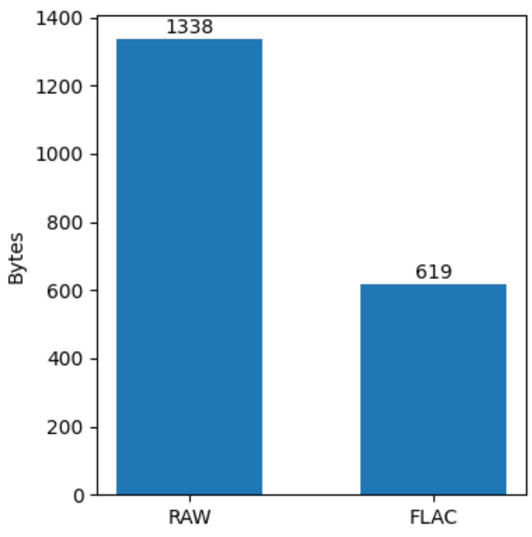

Figure 13: FLAC compression for HPF 2

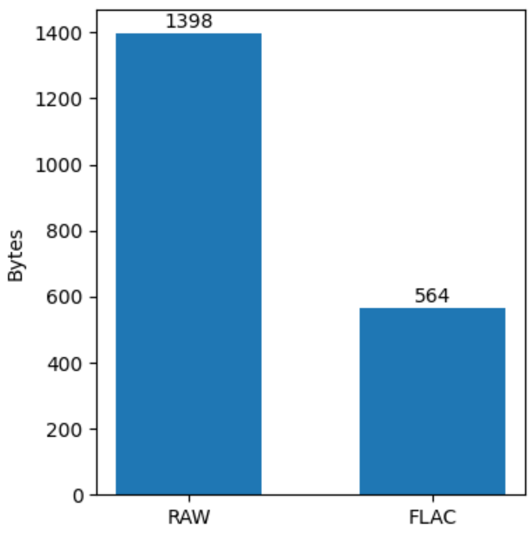

Figure 14: FLAC compression for BPF 2

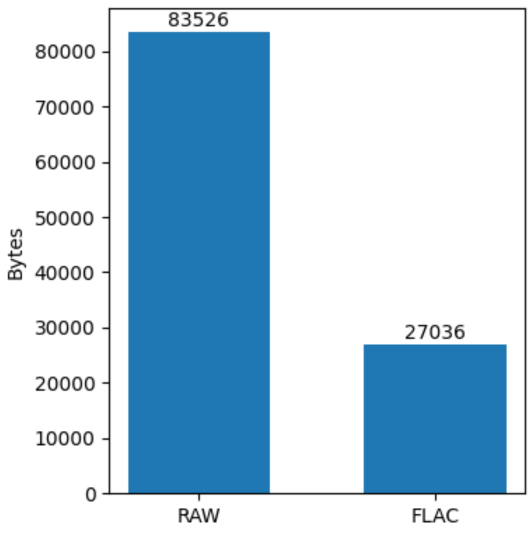

Figure 15: FLAC compression for a long impulse response

This performance can be further improved by removing the FLAC metadata content such as the STREAMINFO and VORBIS_COMMENT blocks, as they are intended for audio file storage/streaming and are not relevant for the purpose of the storage or retrieval of FIR filter coefficients. This would lead to FLAC showing good performance for even a small number of filter coefficients. The ideal compression method derived from FLAC, would employ only the predictor and Rice encoding on a frame by frame basis, and thus would not have any overhead besides the frame header containing the predictor parameters and the encoded residual.

4 Discussion

Our results show that the initial hypothesis, that FLAC compression would be an appropriate fit for compressing FIR filter coefficients owing to the fact that both audio data and filter coefficients show sample to sample correlation, was confirmed. We were also able to demonstrate that FLAC performs consistently better than the current state-of-the-art (successive-difference) method. However, since implementing a FLAC decoder in hardware is more complex than simple successive-difference decoding, trade-offs between performance and storage requirements would need to be considered: it may only be feasible to use FLAC

compression in systems where a very large number of coefficients need to be stored (such as impulse response loaders), and the system can handle the computational overhead of the FLAC decoder. Further, the FLAC algorithm can be stripped-down to having only a few types of predictors and without the metadata information storage, for a more feasible hardware decoder implementation.
For the successive-difference algorithm, the optimised version presented in section 2.3.3 also gives satisfactory results, and is much easier to implement in hardware compared to FLAC. The successive-difference algorithm could be further improved by dividing the array of coefficients into subarrays, processed with the same logic. This way, it is more likely to find parts in which the adjacent coefficients have the same sign and others in which the adjacent coefficients have different sign.

5 Conclusion

Finite Impulse response filters are frequently deployed in embedded systems. Since these devices are resource constrained, optimising storage allocation is crucial. The successive difference method is the current state-of-the-art compression technique for FIR filter coefficients and through our work we were able to benchmark the performances and compare them with the FLAC compression algorithm. Analysing the results obtained from the successive difference method, we were able to provide an optimisation that improves upon the original implementation, making it more flexible to different filter types. However, the FLAC compression technique has shown good performance on all the filter types, and on average gives better compression ratio than even the optimised successive difference method.

References

[1] M. Jabar, "Application of fir filters," Dec. 2021. [Online]. Available: https://www.researchgate.net/publication/356914847_Application_Of_FIR_Filters

[2] N. J. S. Marchon, "Sharp transition fir bandpass filter for processing bioelectric signals," *J. Phys.: Conf. Ser. 1921 012019*, 2021. doi: 10.1088/1742-6596/1921/1/012019. [Online]. Available: https://iopscience.iop.org/article/10.1088/1742-6596/1921/1/012019/pdf

[3] S. Nene and G. N.K. Rangan, "Methods and systems for compression, storage, and generation of digital filter coefficients," *United States Patent US 8,271,567 B2*, Sep. 2012. [Online]. Available: https://patentimages.storage.googleapis.com/c6/a6/66/e71005a71ffe21/US8271567.pdf

[4] J. Coalson, "Flac - format." [Online]. Available: https://xiph.org/flac/format.html

[5] Y. A. R. Tilman Liebchen, "Mpeg-4 als: an emerging standard for lossless audio coding." [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1281489

[6] D. Goehringer, M. D. Santambrogio, J. M.P. Cardoso, and K. Bertels (Eds.), *Reconfigurable Computing: Architectures, Tools, and Applications*. Springer, 2014. ISBN 978-3-319-05959-4

[7] Rockbox Wiki, "Comparison of available decoders across targets." [Online]. Available: https://www.rockbox.org/wiki/CodecPerformanceComparison

[8] T. Robinson, "Shorten: Simple lossless and near-lossless waveform compression," *Technical report CUED/F-INFENG/TR.156*, Dec. 1994. [Online]. Available: https://mi.eng.cam.ac.uk/reports/svr-ftp/auto-pdf/robinson_tr156.pdf

[9] M. Hans and R. W. Schafer, "Lossless compression of digital audio," *IEEE SIGNAL PROCESSING MAGAZINE*, Jul. 2001. [Online]. Available: https://www.eie.polyu.edu.hk/~enyhchan/ce_ac_p1.pdf

[10] F. A. Muin, T. S. Gunawan, M. Kartiwi, and E. M. A. Elsheikh, "A review of lossless audio compression standards and algorithms," *AIP Conference Proceedings 1883, 020006 (2017)*, Sep. 2017. [Online]. Available: https://doi.org/10.1063/1.5002024

A Appendix

A.1 Successive Difference Compression

```python
# Third order successive-difference algorithm implementation

from scipy.io import wavfile
import numpy as np
import math
import matplotlib.pyplot as plt
from bitarray import bitarray


def compute_bits(num):
    if num == 0:
        return 1
    elif num < 0:
        log = math.log(abs(num), 2)
        size = math.ceil(log) + 1
    else:
        # positve
        log = math.log(abs(num), 2)
        if math.ceil(log) == math.floor(log):
            # upper bound is 2**(n-1)-1, so we need 2 bits
            size = math.ceil(log) + 2  # container +1 for extra bit, +1 for sign
    extending
        else:
            size = math.ceil(log) + 1  # container +1 for sign extending
    return size


def compute_diff_false(coeff):
    diff= coeff[i + 1] - coeff[i]
    return diff

def compute_diff_true(coeff):
    diff = coeff[i + 1] + coeff[i]
    return diff



def retrieve_coeff_false(i, first_coeff, diff, retrieve_prec, sign):
    if i == 0:
        retrieve = diff + first_coeff
    else:
        retrieve = diff + retrieve_prec
    return retrieve

def retrieve_coeff_true(i, first_coeff, diff, retrieve_prec, sign):
    if i == 0:
        retrieve = diff - first_coeff
    else:
        retrieve = diff - retrieve_prec
    return retrieve



def choose_operand(coeff):
    counter_same_sign = 0
    counter_different_sign = 0
    sign = False
    DIM = len(coeff)
    for i in range(0, DIM - 1):
        #
        if coeff[i] > 0 and coeff[i + 1] > 0 or coeff[i] < 0 and coeff[i + 1] < 0:
            counter_same_sign += 1
        else:
            counter_different_sign += 1

    if counter_different_sign > counter_same_sign:
        print("ADD")
        sign = True
    else:
        print("SUB")
    print("Same sign counter:", counter_same_sign)
    print("Diff sign counter:", counter_different_sign)
    return sign


# read coefficients from file

samplerate, coeff = wavfile.read(r"C:\Users\leo_m\Downloads\IR.wav")
#coeff = np.fromfile(r"C:\Users\leo_m\Downloads\research\not_opt\BPF_steep_coeff.txt
   ", dtype=np.int16, sep='\n')

DIM = len(coeff)

# print(type(coeff[0]))
# encoding

coeff = np.array(coeff, dtype=np.int32)
diff_1 = np.array([], dtype=np.int32)  # first difference
diff_2 = np.array([], dtype=np.int32)  # second difference
diff_3 = np.array([], dtype=np.int32)  # third difference

# False = - sign
# True  = + sign
sign_1 = False
sign_2 = False
sign_3 = False

counter_same_sign = 0
counter_different_sign = 0

first_coeff_init = coeff[0]
print("DIM", DIM)

sign_1 = choose_operand(coeff)


if sign_1 == False:
    for i in range(0, DIM - 1):
        # diff_1 = np.append(diff_1, coeff[i + 1] - coeff[i])
        # values = compute_diff(coeff)
        diff_1 = np.append(diff_1, compute_diff_false(coeff))
else:
    for i in range(0, DIM - 1):
        # diff_1 = np.append(diff_1, coeff[i + 1] - coeff[i])
        # values = compute_diff(coeff)
        diff_1 = np.append(diff_1, compute_diff_true(coeff))

first_coeff_1 = diff_1[0]

sign_2 = choose_operand(diff_1)

```

```
if sign_2 == False:
    for i in range(0, DIM - 2):
        # diff_1 = np.append(diff_1, coeff[i + 1] - coeff[i])
        # values = compute_diff(coeff)
        diff_2 = np.append(diff_2, compute_diff_false(diff_1))
else:
    for i in range(0, DIM - 2):
        # diff_1 = np.append(diff_1, coeff[i + 1] - coeff[i])
        # values = compute_diff(coeff)
        diff_2 = np.append(diff_2, compute_diff_true(diff_1))

first_coeff_2 = diff_2[0]
sign_3 = choose_operand(diff_2)

if sign_3 == False:
    for i in range(0, DIM - 3):
        # diff_1 = np.append(diff_1, coeff[i + 1] - coeff[i])
        # values = compute_diff(coeff)
        diff_3 = np.append(diff_3, compute_diff_false(diff_2))
else:
    for i in range(0, DIM - 3):
        # diff_1 = np.append(diff_1, coeff[i + 1] - coeff[i])
        # values = compute_diff(coeff)
        diff_3 = np.append(diff_3, compute_diff_true(diff_2))

# decoding


retrieve_1 = np.array([], dtype=np.int32)  # first difference
retrieve_2 = np.array([], dtype=np.int32)  # second difference
retrieve_3 = np.array([], dtype=np.int32)

retrieve_1 = np.append(retrieve_1, first_coeff_init)
retrieve_2 = np.append(retrieve_2, first_coeff_1)
retrieve_3 = np.append(retrieve_3, first_coeff_2)


if sign_3 == False:
    for i in range(0, DIM - 3):
        retrieve_3 = np.append(retrieve_3, retrieve_coeff_false(i, first_coeff_2,
    diff_3[i], retrieve_3[i], sign_3))
else:
    for i in range(0, DIM - 3):
        retrieve_3 = np.append(retrieve_3, retrieve_coeff_true(i, first_coeff_2,
    diff_3[i], retrieve_3[i], sign_3))

if sign_2 == False:
    for i in range(0, DIM - 2):
        retrieve_2 = np.append(retrieve_2, retrieve_coeff_false(i, first_coeff_1,
    retrieve_3[i], retrieve_2[i], sign_2))
else:
    for i in range(0, DIM - 2):
        retrieve_2 = np.append(retrieve_2, retrieve_coeff_true(i, first_coeff_1,
    retrieve_3[i], retrieve_2[i], sign_2))

if sign_1 == False:
    for i in range(0, DIM - 1):
        retrieve_1 = np.append(retrieve_1, retrieve_coeff_false(i, first_coeff_init,
    retrieve_2[i], retrieve_1[i], sign_1))
else:
    for i in range(0, DIM - 1):
        retrieve_1 = np.append(retrieve_1, retrieve_coeff_true(i, first_coeff_init,
    retrieve_2[i], retrieve_1[i], sign_1))
```

```

# check everything is correct
for i in range(0, DIM):
    if retrieve_1[i] != coeff[i]:
        print("error!")

# Compute size of the elements in the array

# print(diff_3)

bits_uncompressed = 0
bits_order_1 = 0
bits_order_2 = 0
bits_order_3 = 0
sum = 0

#bits_uncompressed = DIM * 16

for num in coeff:
        bits_uncompressed = bits_uncompressed + compute_bits(num)

for num_1 in diff_1:
    bits_order_1 = bits_order_1 + compute_bits(num_1)  # container +1 for sign
    extending

bits_order_1 = bits_order_1 + compute_bits(first_coeff_init) + 1 # bug

for num_2 in diff_2:
    bits_order_2 = bits_order_2 + compute_bits(num_2)

bits_order_2 = bits_order_2 + compute_bits(first_coeff_init) + compute_bits(
    first_coeff_1) + 2

for num_3 in diff_3:
    bits_order_3 = bits_order_3 + compute_bits(num_3)

bits_order_3 = bits_order_3 + compute_bits(first_coeff_init) + compute_bits(
    first_coeff_1) + \
                compute_bits(first_coeff_2) + 3

print("\nCompression rate:\n")
print("1st order:", (1 - bits_order_1 / bits_uncompressed) * 100)
print("2st order:", (1 - bits_order_2 / bits_uncompressed) * 100)
print("3st order:", (1 - bits_order_3 / bits_uncompressed) * 100)


'''
bit_uncompressed = math.ceil((math.ceil(math.log(np.max(coeff), 2)) * len(coeff)) /
    8)
bits_order_1 = math.ceil((math.ceil(math.log(np.max(diff_1), 2)) * len(diff_1)) / 8)
    + 1
# +1: to store the 1st coefficient
bits_order_2 = math.ceil((math.ceil(math.log(np.max(diff_2), 2)) * len(diff_2)) / 8)
    + 2
# +2: store the 1st and 2nd coefficient
bits_order_3 = math.ceil((math.ceil(math.log(np.max(diff_3), 2)) * len(diff_3)) / 8)
    + 3
# +3: store the 1st, 2nd and 3rd coefficient

'''

# Plot the data

```

```
data = [bits_uncompressed, bits_order_1, bits_order_2, bits_order_3]

labels = ['Uncompressed', 'First Order', 'Second Order', 'Third Order']

x = np.arange(len(labels))  # the label locations
width = 0.8  # the width of the bars

fig, ax = plt.subplots(figsize=(5, 4))
rects1 = ax.bar(x, data, width)

ax.set_ylabel('Bits')
ax.set_xticks(x, labels)
ax.legend()

ax.bar_label(rects1, padding=0.8)

fig.tight_layout()

plt.show()
```

A.2 FLAC Compression

```python
import queue
import numpy as np
import pyflac
import matplotlib.pyplot as plt
from scipy.io import wavfile
import math

idx = 0
total_bytes = 0
rate = 44100
data_queue = queue.SimpleQueue()
tmp = np.array

def compute_bits(num):
    if num == 0:
        return 1
    elif num < 0:
        log = math.log(abs(num), 2)
        size = math.ceil(log) + 1
    else:
        # positve
        log = math.log(abs(num), 2)
        if math.ceil(log) == math.floor(log):
            # upper bound is 2**(n-1)-1, so we need 2 bits
            size = math.ceil(log) + 2  # container +1 for extra bit, +1 for sign
    extending
        else:
            size = math.ceil(log) + 1  # container +1 for sign extending
    return size

def write_callback(buf: bytes, num_bytes: int, num_samples: int, frame_num: int):
    global total_bytes
    total_bytes += num_bytes
    data_queue.put(buf)
    print('Samples ', frame_num)
    print('\n', buf)

def read_callback(decoded_data: np.array, sample_rate: int,
                  num_channels: int, num_samples: int):
    global idx
    global tmp
    idx += num_samples
    tmp = decoded_data

data = np.fromfile('files/HPF_steep_coeff.txt', dtype=np.int16, sep='\n')
#samplerate, data = wavfile.read('files/ir.wav')
temp = 0

for i in data:
    temp += compute_bits(i)

print("BITS: ", temp)


encoder = pyflac.StreamEncoder(rate, write_callback, blocksize = 0, verify=True)
encoder.process(data)
encoder.finish()

decoder = pyflac.StreamDecoder(read_callback)
while not data_queue.empty():
    decoder.process(data_queue.get())
decoder.finish()

tmp.shape = (1, tmp.size)
data.shape = (1, data.size)
if(np.array_equal(tmp, data)):
    print('Decoded value OK')

print('\nUncompressed: ' +  str(data.nbytes) + ' bytes\n')
print('Compressed: ' +  str(total_bytes) + ' bytes\n')
print('Ratio: {ratio:.2f}%'.format(ratio= (1-(total_bytes / data.nbytes) )* 100))

names = ['RAW', 'FLAC']
values = [data.nbytes, total_bytes]

x = [0,1]
width = 0.6 # the width of the bars
fig, ax = plt.subplots(figsize=(4, 4))
rects1 = ax.bar(x, values, width)

ax.set_ylabel('Bytes')
ax.set_xticks(x, names)
ax.legend()

ax.bar_label(rects1, padding=0.7)

fig.tight_layout()

plt.show()
```