Accelerating multi-channel filtering of audio signal on ARM processors

Jose A. Belloch · Fran J. Alventosa · Pedro Alonso · Enrique S. Quintana-Ortí · Antonio M. Vidal

Published online: 1 March 2016
© Springer Science+Business Media New York 2016

Abstract Tablets and smart phones are nowadays equipped with low-power processor architectures such as the ARMv7 and the ARMv8 series. These processors integrate powerful SIMD units to exploit the intrinsic data-parallelism of most media and signal processing applications. In audio signal processing, there exist multiple problems that require filtering operations such as equalizations or signal synthesizers, among others. Most of these applications can be efficiently executed today on mobile devices by leveraging the processor SIMD unit. In this paper, we target the implementation of multi-channel filtering of audio signals on ARM architectures. To this end, we consider two common audio filter structures: FIR and IIR. The latter is analyzed in two different forms: direct form I and parallel form. Our results show that the SIMD-accelerated implementation increases the processing speed by a factor of 4× with respect to the original code, and our hand-tuned SIMD implementation outperforms the auto-vectorized code by a factor of 2×. These results allow us to deal in real time with...
multi-channel systems composed of 260 FIR filters with 256 coefficients, or 125 IIR filters with 256 coefficients, of INT16 data type.

Keywords Low-power processors · ARMv7 and ARM® Cortex-A15 · NEON® Intrinsics · Audio processing

1 Introduction

In the era of multimedia, smart phones and tablets, low-power processors play an important role for a myriad of applications. The ARM® Cortex-A15 [1] is an implementation of the ARMv7 architecture that was conceived as an embedded processor for smartphones, among other mobile appliances. In particular, each core of this processor integrates a 128-bit SIMD (single-instruction multiple-data) engine, called NEON® that may significantly increase performance as well as reduce energy consumption for multimedia and signal processing applications. From the programmer’s perspective, data-parallelism can be exploited by means of specific SIMD instructions, or alternatively via vector NEON intrinsics.

There exist multiple applications in audio signal processing that involve filtering operations, such as equalization [2], audio synthesizers [3–5], multi-channel signal processing [6–8], and headphone-based spatial sound [9,10]. Although these applications may meaningfully improve the listening experience [11], their implementation on a mobile devices poses a challenging task.

In this paper, we focus on the filtering operation using two common filter structures: FIR (finite impulse response) and IIR (infinite impulse response). Regarding the IIR structure, we target two different forms: direct form I (IIRI) and parallel form (PIIR). The PIIR structure is a parallel version of IIR composed of multiple second-order sections [12] developed by Bank in [13]. The proposed implementations exploit the NEON® engine capabilities integrated in the ARM® Cortex-A15 core to extract data-parallelism, delivering a remarkable increase in the number of filter operations that can be handled. We note that our work focuses only on improving the computational performance and does not assess the viability of the multi-channel filtering inside a more specific application. In practice, most high-quality audio applications require 32-bit integer or floating-point processing, due to the larger quantization noise presented by 16-bit filtering.

ARM processors have been previously applied in a number of audio/video signal processing scenarios. In [14], the authors accelerate various image processing algorithms using an ARM architecture, and an image processing application is optimized using NEON intrinsics in [15]. ARM-based platforms have also been used in [16] to accelerate the Audio Video coding Standard (AVS) and audio processing with IIR filters in [17].

Compared with previous work, we target multi-channel filtering using different filter structures and different data types, including arithmetic with integers of 16 bit. Moreover, we evaluate the performance when the multi-channel filtering uses the four cores of the target ARM processor.
The rest of the paper is structured as follows. In Sect. 2, we review the following filter structures: FIR, IRRI, and PIIR. Section 3 presents our optimized implementation of the filter structures using NEON intrinsics. The performance of the resulting codes is evaluated in Sect. 4. Finally, Sect. 5 summarizes our work into a few concluding remarks.

2 Filter structures

Filters are a basic component of digital signal processing and telecommunication systems. In this paper, we focus on linear time-invariant (LTI) filters, where the output is a linear combination of the input and the coefficients do not vary with time.

In the time-domain, the input–output relationship of a discrete-time linear filter is given by the linear difference equation:

\[ y(m) = \sum_{k=1}^{N} a_k y(m - k) + \sum_{k=0}^{M} b_k x(m - k), \]

where \(a_k\) and \(b_k\) are the filter coefficients. Thus, the output \(y(m)\) is a linear combination of the previous \(N\) output samples, the current input sample \(x(m)\), and the previous \(M\) input samples. For a time-invariant filter, the coefficients \(a_k\) and \(b_k\) are constants determined with a concrete purpose, such as applying a specific equalization, and obtaining spatial audio effect. Different aspects can be especially relevant when a filter is designed such as stability, computational cost or possibility of arithmetic overflow, among others. In any case, these requirements are always determined by the application.

Depending on the existence of a feedback, we can distinguish between two types of filter structures: FIR and IIR. We next review these two.

2.1 FIR filter

FIR has no feedback, and its input–output relation is given by

\[ y(m) = \sum_{k=0}^{M} b_k x(m - k). \]

Thus, the output \(y(m)\) of an FIR filter is a function of the input signal \(x(m)\) only, as shown in Fig. 1. The response of such a filter to an impulse consists of a finite sequence of \(M + 1\) samples, where \(M\) is the filter order. Note that each unit delay is a \(z^{-1}\) operator in Z-transform notation [12].

FIR filters are totally stable, since the output is a sum of a finite number of finite multiples of the input values. However, they typically require more computation in comparison with IIR filters conceived with a similar purpose [12].
2.2 IIR filter

IIR filters present a feedback loop with a recursive structure, as shown in Fig. 2. Filters with an IIR structure are characterized because once they have been excited by an impulse, the output persists forever. IIR filters use a small number of coefficients in comparison with FIR filters for the same objective. This implies less storage requirements and faster calculation. Therefore, an IIR filter can become unstable [12]. The feedback loop must be controlled since otherwise the output signal can increase infinitely. Moreover, FIR filters have the additional advantage of being able to implement arbitrary phase responses as opposed to IIR filters. This is beneficial in linear-phase filtering, and non-minimum phase loudspeaker equalization, for example.

There exist different structures for IIR filters, depending on the stability and arithmetic overflow requirements. In this paper, we first analyze the generalized structure of IIR filters that is shown in Fig. 2. This structure is known in the literature as “direct form I”, and is denoted in this paper as IIRI. This generalized case corresponds to the same equation given in (1).

High-order IIRI filters (high number of coefficients) can easily become unstable due to the effects of coefficient quantization. These effects can be reduced if the IIRI filter is decomposed into multiple second-order sections [18], which can be executed in parallel yielding shorter execution time, at the expense of higher quantization noise due to the multiple accumulators. In particular, in this work we tackle in this work the parallel form proposed by Bank in [13], which is used in multiple audio applications such as [2,7]. This specific parallel structure, unlike the general structure that can be
found in [12], presents only one coefficient $d_0$ without delay as shown in the left-hand side of Fig. 3. On the right-hand side, it is important to remark that all coefficients $b_2$ equal to zero for the second-order sections. This parallel structure is denoted in this paper as PIIR, and follows the equation:

$$H(z^{-1}) = \sum_{k=1}^{K} b_{k,0} + b_{k,1}z^{-1} + a_{k,1}z^{-1} + a_{k,2}z^{-2} + d_0,$$

where $K$ is the number of second-order sections. The filter structure and the second-order section are both depicted on the left- and right-hand sides of Fig. 3, respectively.

### 3 System setup and implementation

The ARMv7 architecture and Instruction Set Architecture (ISA) for the Cortex-A15 processor include support for a collection of SIMD floating-point instructions that can apply the same operation to multiple packed elements of the same type and size in parallel. To support this functionality, the Cortex-A15 processor comprises a number of 128-bit SIMD registers, each of which can store up to two 64-bit (double-precision) floats, two 64-bit (long) integers, four 32-bit (single-precision) floats, four 32-bit integers, eight 16-bit integers, or sixteen 8-bit integers.

The ARMv7 NEON intrinsics support all the functionality of the NEON ISA, including the definition of vector data types [19] to place data into vector registers. These types are named as "type"x"number_of_lines"_t, for example: "int16x4_t", "int32x4_t" or "flo32x4_t".

We implemented the FIR, IIRI and PIIR filter structures on the quad-core ARM Cortex-A15 (running at 2.32 GHz) that is part of the Jetson TK1 DevKit. In a real-time application, the audio card receives and produces audio samples each $t_{\text{buff}}$ sec. We set the buffer size to 1024 samples per channel, with a sampling frequency of $f_s = 44.1$ kHz. This means that the audio buffers are filled every 23.22 ms, i.e., $t_{\text{buff}} = 23.22$ ms. This time is important since it determines the threshold for an application that works in real time. For the tests, we consider that the audio samples can be composed of three different data types: 16-bit integers, 32-bit integers, and 32-bit float samples.
3.1 Implementation

In our implementation we use two key vector structures, for the input/output of the filtering operation. Concretely, $X$ contains the audio samples of a current input data; and $Y$ contains the output data after the filtering has been applied. In addition, we use a vector $B$ for the values of the coefficients that are element-wise multiplied by vector $X$. In the IIR filter types, there is a vector $A$ which stores the values of the required coefficients to carry out the feedback loop. For the implementation of the PIIR filter there are four vectors, denoted as $A_1$, $A_2$, $B_1$, and $B_2$, and two other auxiliary vectors, $V_1$ and $V_2$, to store intermediate results.
SIMD instructions are appreciably efficient when the entries of arrays $X$, $Y$, $A$, $B$, $A1$, $A2$, $B1$, $B2$, $V1$ and $V2$ are stored in consecutive memory positions so that the operations can be performed on 2 or 4 elements concurrently. Note that the implementation based on 16-bit integer data type uses vectors composed of int16x4_t elements. For the 32-bit float and the 32-bit integer data types, we use float32x4_t and int32x4_t, respectively. In addition, vectors $A$ and $B$ are rearranged to optimize performance. Part of this reorganizes the array contents to access data in increasing order so that SIMD instructions can be properly leveraged. An important aspect in the implementation lies in the registers that are used to store intermediate results when operating with integers. For 16-bit and 32-bit integers, we use intermediate registers of 32-bit integers (int32x4_t) and 64-bit integers (int64x2_t), respectively. This is necessary to avoid overflow.

Prior to multiplication, the bits that configure the coefficient values must be shifted to the left (coefficient value is multiplied by a scale factor [17] to cast the real number into an integer). After the multiplication, the same shift is undone. This operation aims to improve the accuracy of the multiplication. Algorithms 1, 2, and 3 detail our hand-tuned vectorized implementations using NEON intrinsics of the three different filter structures: FIR, IIRI, and PIIR, respectively. Table 1 briefly summarizes and describes the set of NEON instructions that were utilized for the implementations.

### 4 Performance evaluation

We have evaluated our vectorized implementations based on NEON® instructions. Implementations were compiled with the GNU compiler with flags -O3, -mfpu=neon, and -march=armv7. Figure 4 shows the execution time required by the three filter structures: FIR, IIRI, and PIIR. We used three different data types: 16-bit (INT16) and 32-bit (INT32) integer data, and 32-bit floating-point (FL032). Tests were executed using two different dimensions for $M$ and $N$: 52 and 256 coeffi-

---

**Algorithm 3** Implementation of PIIR filter processing using 16-bit integers

**Input:** $X$, $B1$, $A1$, $B2$, $A2$

**Output:** $Y$

1: int32x4_t acum, int16x4 v0; int32x2_t res; //Auxiliary variables
2: for i=0 to nsize: //Iterates through the input and output data
3:   Y[i] = 0; acum = vmovn_s32(0); //Initialize auxiliary variables
4: for j=0 to absize: //Iterates through the vectors of coefficients
5:   v0 = vmovn_s16(x[i]); //Load the same value (x[i]) in all positions
6:   v0 = vmls_s16(v0, a1[j], v1[j]); //multiply a1 & v1 & subtract to v0
7:   v0 = vmls_s16(v0, a2[j], v2[j]); //multiply a2 & v2 & subtract to v0
8:   acum = vmlal_s16(acum, b1[j], v0); //multiply b1 & v0 & accum. to acum
9:   acum = vmlal_s16(acum, b2[j], v1[j]); //multiply b2 & v1 & accum. to acum
10:  vst1_s16(&v2[j],v1[j]); //Copy the value of v1[j] to v2[j]
11:  vst1_s16(&v1[j],v0); //Copy the value of v0 to v1[j]
12: end for
13: res = vpadd_s32(vget_high_s32(acum),vget_low_s32(acum)); //Add pairwise
14: res = vrsrrn_s32(res, DESP); //Shift values of inter, DESP positions
15: for j=0 to 1: y[i] += (int16_t)vget_lane_s32(res, j); //Store result
16: y[i] += FIR*x[i] //FIR is a constant value

© Springer
Table 1 Instructions NEON employed with a brief description

| Instruction                  | Description                                      |
|------------------------------|--------------------------------------------------|
| vmovq_n_s32(value)           | Initialize int32x4_t vector registers             |
| vmovq_n_s64(value)           | Initialize int64x2_t vector registers             |
| vmovq_n_f32(value)           | Initialize flo32x4_t vector registers             |
| vmovq_n_s16(value)           | Initialize int16x4_t vector registers             |
| vmovq_n_s32(value)           | Initialize int32x2_t vector registers             |
| vld1_s16(*data)              | Load int16x4_t vector registers                   |
| vld1_s32(*data)              | Load int32x4_t vector registers                   |
| vld1q_f32(*data)             | Load flo32x4_t vector registers                   |
| vmls_s16(dest,data1,data2)   | Vector multiply subtract, int16x4_t dest. registers |
| vmls_s32(dest,data1,data2)   | Vector multiply subtract, int32x2_t dest. registers |
| vmlsq_f32(dest,data1,data2)  | Vector multiply subtract, flo32x4_t dest. registers |
| vmlal_s16(dest,data1,data2)  | Vector multiply accumulate long, int32x4_t dest. registers |
| vmlal_s32(dest,data1,data2)  | Vector multiply accumulate long, int64x2_t dest. registers |
| vmlaq_f32(dest,data1,data2)  | Vector multiply accumulate, flo32x4_t dest. registers |
| vst1_s16(dest,source)        | Store a single vector into memory, *int16_t[4] dest. registers |
| vst1_s32(dest,source)        | Store a single vector int32_t[4] dest. registers |
| vst1q_f32(dest,source)       | Store a single vector into memory, *flo32_t[4] dest. registers |
| vget_high_s32(reg)           | Splitting vector registers, int32x2_t return vector registers |
| vget_high_f32(reg)           | Splitting vector registers, flo32x2_t return vector registers |
| vget_low_s32(reg)            | Splitting vector registers, int32x2_t return vector registers |
| vget_low_f32(reg)            | Splitting vector registers, flo32x2_t return vector registers |
| vpadd_s32(data1,data2)       | Pairwise add, int32x2_t dest. vector registers    |
| vpadd_f32(data1,data2)       | Pairwise add, flo32x2_t dest. vector registers    |
| vget_lane_s32(reg,pos)       | Extract lanes from a vector, int32_t dest. registers |
| vget_lane_s64(reg,pos)       | Extract lanes from a vector, int64_t dest. registers |
| vget_lane_f32(reg,pos)       | Extract lanes from a vector, flo32_t dest. registers |

coefficients. As shown, the best performance is obtained for data types INT16 and FLO32, except for the case of FIR filters, where data type INT32 outperforms the remaining alternatives. This occurs because of the lack of a feedback loop in the FIR structure.

We next compare our hand-tuned implementation based on intrinsic NEON® functions, for two data types INT16 and FLO32, with respect to an implementation that
was compiled with auto-vectorized options [20] (flag `-ftree-vectorize` of the compiler). The speed-up of our hand-tuned implementation is shown in Fig. 5. The results emphasize the large benefit that can be obtained with the 16-bit integer data version compared with the 32-bit float data type. Moreover, our proposed implementation offers higher performance than the auto-vectorized code, especially when the data type INT16 is used. For the FIR structure, the speed-ups are above $5 \times$ and $7 \times$ when the filter is composed of 52 and 256 coefficients, respectively.

We have executed all implementations in parallel by launching multiple threads through OpenMP directives. Since the filtering operations are totally independent, they can be distributed among the available cores. Figure 6 shows the scalability of the implementations for filters composed of an IIRI structure.

Let us define $t_{\text{proc}}$ as the processing time elapsed since the input buffer is available until the output buffer is totally processed. Assuming that we can execute $F$ filtering operations simultaneously, $t_{\text{proc}}$ now accounts for the processing time elapsed since the $F$ input buffers are available until the $F$ output buffers are totally processed.
The filtering operation works within the real-time threshold provided $t_{\text{proc}} < t_{\text{buff}}$. The objective of the following experiment is to assess the largest number of filtering operations, $F_{\text{max}}$, that can be computed in real time on the target architecture using the proposed implementations. To determine this factor, we executed the codes for an increasing number of filters, and compared the execution time $t_{\text{proc}}$ with the threshold $t_{\text{buff}}$.
Audio signal multi-channel filtering…

Figure 7 illustrates the evolution of $t_{\text{proc}}$ as the number of filtering operations increases for the data types INT16 and FLO32. We can observe that the maximum number of IIRI filters with 256 coefficients that can be executed under real-time conditions, using the NEON® intrinsics, is 80 for FLO32, and 125 for INT16. This corresponds to an increment of $4 \times$ in comparison with the implementation that does not use the NEON® intrinsics.

Regarding the FIR filters, they present a better vectorization since they do not present a feedback loop. As a result, the ARM® architecture is able to manage more filtering operations with the FIR structure than with the IIR. Specifically, for FIR filters with 256 coefficients, 170 filtering operations can be executed in real time for FLO32 and 260 for INT16. This represents again a speed-up $4 \times$ attained with the use of the NEON® intrinsics.

5 Conclusions

In this paper, we have introduced efficient implementations of the FIR, IIRI, and PIIR filter structures that leverage the NEON® intrinsics available on the ARM® Cortex-A15. The results show that our hand-tuned vectorized code outperforms the counterpart that is compiled with auto-vectorized options by a large margin ($4 \times$). We also have exploited the four ARM cores via OpenMP improving the performance by a factor which is close to $4 \times$ for all the filter structures. As a result, our implementation is able
to handle in real time up to 125 IIR filters and 260 FIR filters with 256 coefficients concurrently for a data type of INT16.

**Acknowledgements**  The researchers from Universitat Jaume I are supported by the CICYT projects TIN2014-53495-R and TIN2011-23283 of the Ministerio de Economía y Competitividad and FEDER. The authors from the Universitat Politècnica de València are supported by projects TEC2015-67387-C4-1-R and PROMETEOII/2014/003. This work was also supported from the European Union FEDER (CAPAP-H5 network TIN2014-53522-REDT).

**References**

1. ARM NEON. [http://www.arm.com/](http://www.arm.com/). Accessed 23 Feb 2015
2. Rämo J, Välimäki V, Bank B (2014) High-precision parallel graphic equalizer. IEEE Trans Audio Speech Lang Process 22:1894–1904
3. Mathews MV, Miller JE, Moore FR, Pierce JR, Risset JC (1969) The technology of computer music. MIT Press, Cambridge, Mass
4. Risset JC (1985) Computer music experiments 185. Comput Music J 22:11–18
5. Puckette M (2007) The theory and technique of electronic music, Word Scientific Publishing ISBN-13: 978–9812700773
6. Savioja L, Välimäki V, Smith JO (2011) Audio signal processing using graphics processing units. J Audio Eng Soc 59:3–19
7. Belloch JA, Bank B, Savioja L, Gonzalez A, Välimäki V (2014) Multi-channel IIR filtering of audio signals using a GPU. In: Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing (ICASSP-14), pp 6692–6696
8. Belloch JA, Gonzalez A, Martinez-Zaldívar FJ, Vidal AM (2013) Multichannel massive audio processing for a generalized crosstalk cancellation and equalization application using GPUs. Integr Comput Aided Eng 20:169–182
9. Algazi V, Duda R (2011) Headphone-based spatial sound. IEEE Signal Process Mag 28:33–42
10. Belloch JA, Ferrer M, Gonzalez A, Martinez-Zaldívar FJ, Vidal AM (2013) Headphone-based virtual spatialization of sound with a GPU accelerator. J Audio Eng Soc 61:546–556
11. Huang Y, Chen J, Benesty J (2011) Immerse audio schemes. IEEE Signal Process Mag 28:20–32
12. Oppenheim AV, Willsky AS, Hamid S (1997) Signals and systems, processing series, 2nd edn. Prentice Hall, Upper Saddle River
13. Bank B (2008) Perceptually motivated audio equalization using fixed-pole parallel second-order filters. IEEE Signal Process Lett 15:477–480
14. Mitra G, Johnston B, Rendell AP, McCreath E, Zhou J (2013) Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel Platforms. In: IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), pp 1107–1116
15. Welch E, Patru D, Saber E, Bengtson K (2012) A study of the use of SIMD instructions for two image processing algorithms. Western New York Image Processing Workshop (WNYIPW), pp 21–24
16. Wang R, Wan J, Wang W, Wang Z, Dong S, Gao W (2013) High definition IEEE AVS decoder on ARM NEON platform. In: 20th IEEE International Conference on Image Processing (ICIP), pp 1524–1527
17. Holgersson SB (2012) Optimising IIR filters using ARM NEON, Master Thesis of University of Denmark
18. Rabiner LR, Gold B (1975) Theory and application of digital signal processing. Prentice-Hall, Englewood Cliffs
19. ARM NEON intrinsics. [http://gcc.gnu.org/onlinedocs/gcc-4.4.1/gcc/ARM-NEON-Intrinsics.html](http://gcc.gnu.org/onlinedocs/gcc-4.4.1/gcc/ARM-NEON-Intrinsics.html). Accessed 12 July 2015
20. ARM NEON auto-vectorization. [http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html](http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html). Accessed 22 July 2015