VECTORIZATION OF BINAURAL SOUND VIRTUALIZATION ON THE ARM CORTEX-A15 ARCHITECTURE

Jose A. Belloch[⋆], *Alberto González*[†], *Francisco D. Igual*[‡], *Rafael Mayo*[⋆], *Enrique S. Quintana-Ortí*[⋆]

[⋆]Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I, Castellón, Spain
[†]iTeAm, Universitat Politècnica de València, Valencia, Spain
[‡]Depto. de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, Madrid, Spain

ABSTRACT

Today's mobile devices are equipped with low power processors featuring SIMD (single-instruction, multiple-data) floating-point units which can operate with multiple data units concurrently. This is the case, e.g., of the ARMv7 architecture, which integrates the (NEON) *Advanced SIMD extension*, a combined 64- and 128-bit SIMD instruction set for standardized acceleration of media and signal processing applications. In this paper we target the efficient implementation of binaural sound virtualization, a heavy-duty audio processing application that can eventually require 16 convolutions to synthesize a virtual sound source. For this application, we describe a data reorganization that allows to exploit the 128-bit *NEON intrinsics* of an ARM Cortex-A15 core. As a result, our new SIMD-accelerated implementation is capable of reproducing up to 60 sound sources under real-time conditions, compared with the 40 sound sources that can be handled by the original code.

Index Terms— Audio Processing, Spatial Sound, Low Power Processors, ARMv7 and ARM Cortex-A15, NEON Intrinsics.

1. INTRODUCTION

In the era of multimedia, smart phones and tablets, low power processors play a crucial role for a large volume of applications that is rapidly growing. Among these specialized devices, the ARM Cortex-A15 [1] is an implementation of ARMv7 architecture that was conceived as an embedded processor for smart phones, among other mobile appliances. In particular, each core of this processor integrates a 128-bit SIMD (single-instruction, multiple-data) engine, called NEON, that can significantly increase performance and 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 with vector intrinsics to increase ease of programming.

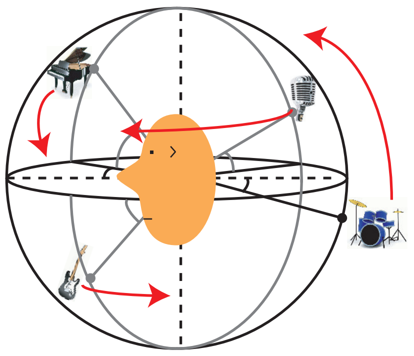

Fig. 1. Binaural sound virtualization intends to reproduce moving sound sources.

One important application in signal processing is *binaural sound virtualization*, where the goal is to leverage a spatial audio system, based on headphones, to allow a listener perceive the virtual position of a sound source [2]. This effect is obtained by processing sound samples through a collection of special filters that shape the sound with spatial information. In the frequency domain, these filters are known as *Head-Related Transfer Functions* (HRTFs), and the response of HRTFs describes how a sound wave is affected by properties of the individual's body (e.g. pinna, head, shoulders, neck and torso) before the sound reaches the listener's eardrum [3]. From the computational perspective, the problem that underlies a headphone-based application quickly grows in cost with the number of sound sources that have to be handled (i.e., reproduced and moved) in real time while avoiding acoustic artifacts, see Figure 1.

In this paper we introduce a tailored implementation of binaural sound virtualization that exploits the NEON engine capabilities integrated in the ARM Cortex-A15 core to extract data parallelism, delivering a remarkable increase in the number of moving sound sources that can be handled. The key to this development is a reorganization/duplication of the audio data in memory to leverage the SIMD capabilities of the

ARM-NEON floating-point units, by appropriately employing vector intrinsics in the code.

The rest of the paper is structured as follows. In Section 2 we briefly discuss some related work; in Section 3 we introduce the spatial sound application; and in Section 4 we shortly review the ARM NEON intrinsics. Next, in Section 5 we present our specialized implementation of the binaural audio virtualization code and evaluate its performance; and Section 6 summarizes our work into a few concluding remarks.

2. RELATED WORK

ARM processors have been previously applied in a number of audio/video signal processing scenarios. In [4], the authors accelerate various image processing algorithms using an ARM architecture, and image processing is optimized using NEON intrinsics in [5]. ARM-based platforms have also been used in [6] to accelerate the Audio Video coding Standard (AVS) and different types of IIR filters have been implemented for audio filtering using NEON extensions in [7].

Our work differs in that none of the previous efforts has tackled filtering algorithms in the frequency domain, which requires Fourier transforms as well as complex arithmetic.

3. SPATIAL AUDIO ON HEADPHONES

Spatial effects can be achieved by convolving natural monophonic sounds that are recorded in an anechoic environment with a pair of filters that add spatial information from specific positions in the space to the audio wave.

For headphones, one filter per ear specifies each virtual position, known as *head-related impulse response* (HRIR) filter in the time domain and HRTF filter in the frequency domain. The HRIRs corresponding to position (θ, ϕ, r) in the time domain are denoted as $\mathbf{h}_\mathrm{r}(\theta, \phi, r)$ and $\mathbf{h}_\mathrm{l}(\theta, \phi, r)$ for the right and left ear, respectively.

HRTFs are usually obtained via measurement combined with extrapolation or numerical simulation [8]. There are multiple public samples of HRTFs or HRIRs. In our case, we leverage the HRIR measures from [9]. This HRIR database has azimuth and elevation resolutions, denoted by $\Delta\theta$ and $\Delta\phi$ respectively, representing the minimum separation in degrees between two positions of the database in azimuth and elevation. For our HRIR database, the resolution for both azimuth and elevation is 15°, and the distance of the sound source to the center of the head is fixed to r=1.95 m. Moreover all HRIR filters are windowed to a length of 512 coefficients.

Let us employ $\mathbf{x}_\mathrm{buff}$ to denote an input-data buffer composed of L audio samples from a sound source x. Given a system composed of M sources, the input-data buffer $\mathbf{x}_{\mathrm{buff}_i}$ represents the buffer of L samples from source $i \in [0, M-1]$. The output-data buffers both for the left and right ears, $\mathbf{y}_\mathrm{buff}$, is then given in the time domain by

$$\mathbf{y}_\mathrm{buff} = \sum_{i=0}^{M-1} (\mathbf{h}_\mathrm{l}(\theta_i, \phi_i) * \mathbf{x}_{\mathrm{buff}i}). \tag{1}$$

where $*$ denotes the convolution operator.

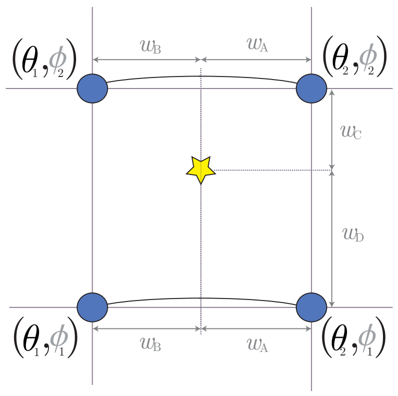

Fig. 2. The star represents the position intended to be synthesized in the elevation plane ϕ and in the azimuth plane θ . This position is synthesized by combining the nearby azimuth and elevation positions using the appropriate weighted factors.

3.1. Virtualization of sound source Movements

The number of filters in the database constrains the virtual positions which can be rendered. One complex problem is the synthesis of sound for virtual positions that do not belong to the database. An additional difficulty occurs when sound source moves, which in practice requires the application of a new filter (i.e., a different HRIR). In particular, if the switch between HRIRs is not properly implemented, this may yield multiple audio clipping effects [10].

In order to address these problems, we carry out 16 convolutions for each moving sound source, as discussed in [11]. Concretely, two filters correspond to the left/right channels, and four more are required to synthesize positions that are not available in the database. Equation 2 and Figure 2 shows how the interpolation is implemented. There, $\{w_A, w_B, w_C, w_D\}$ correspond to weighted factors.

$$\begin{aligned} \mathbf{y}_{\mathrm{buff}_i}(\theta_S, \phi_S) &= w_D \cdot w_B \cdot \mathbf{y}_{\mathrm{buff}_i}(\theta_1, \phi_1) \\ &+ w_D \cdot w_A \cdot \mathbf{y}_{\mathrm{buff}_i}(\theta_2, \phi_1) \\ &+ w_C \cdot w_B \cdot \mathbf{y}_{\mathrm{buff}_i}(\theta_1, \phi_2) \\ &+ w_C \cdot w_A \cdot \mathbf{y}_{\mathrm{buff}_i}(\theta_2, \phi_2). \end{aligned} \tag{2}$$

In addition, two more filtering are necessary for the virtualization of source movement, which is carried out by smoothly varying the virtual positions of the source over time. For example, assume the sound source x_i moves from position A: (θ_{SA}, ϕ_{SA}) to position B: (θ_{SB}, ϕ_{SB}). We then apply a fading, which is a gradual increase in the sound filtered by position B while the sound filtered by position A alternates in the same proportion. To this end, the current input-data buffer $\mathbf{x}_{\text{buff}_i}$ must be computed for both positions, and then the result of both computations must be multiplied element-wise by two fading vectors, say $\mathbf{f}$ and $\mathbf{g}$. Finally, the output-data buffer $\mathbf{y}_{\text{buff}_i}$ is obtained by summing the results from the previous multiplications element-wise; i.e.,

$$\begin{aligned} \mathbf{y}_{\text{buff}i}(\theta_S, \phi_S) &= ((\mathbf{y}_{\text{buff}_i}(\theta_{SB}, \phi_{SB}) \otimes \mathbf{f}) \\ &\oplus \; ((\mathbf{y}_{\text{buff}_i}(\theta_{SA}, \phi_{SA}) \otimes \mathbf{g}) \end{aligned} \tag{3}$$

where $\otimes$ and $\oplus$ respectively represent the element-wise multiplication and addition operators.

4. EXPLORING THE ARM CORTEX-A15

The ARMv7 architecture and instruction set architecture (ISA), implemented in the Cortex-A15 processor among others, include support for a set of SIMD floating-point instructions that can apply the same operation on multiple packed elements of the same type and size in parallel. In order 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) *float* elements, four 32-bit (single-precision) *float* elements, or eight 16-bit *integers*. Each piece of data stored in a SIMD register is usually referred to as a *lane*. Typical SIMD instructions supported by ARM processors include arithmetic and logical operations (e.g. addition, multiplication, multiply-accumulate, multiply-subtract, subtraction, comparison, absolute difference) and data movement instructions (e.g., load lanes from memory into a SIMD register, and store lanes from a SIMD register into memory).

The implementation of the SIMD technology in the ARMv7 architecture comes in the form of a separate vector floating-point unit, called NEON. The vector register set in the NEON engine and the general-purpose register set in the ARM core are independent. In order to increase ease of programming, SIMD operations can be cast into intrinsic functions that are directly translated by the compiler into NEON instructions, operating in parallel over different lanes of a SIMD register.

The ARMv7 NEON intrinsics support all the functionality of the NEON instruction set, including the definition of vector data types [12] to place data into vector registers. These types are named as *"type""size"x"number of lanes"_t*. Some datatype examples include `int8x8_t`, `int8x16_t`, `int16x4_t`, `int32x4_t`, `int64x2_t`, `float32x_t`, `float32x2_t`, and `float32x4_t`.

5. IMPLEMENTATION AND PERFORMANCE

The convolution operations in our implementation of the binaural audio virtualization are carried out by using the overlap-save technique with a 50% overlap in the frequency domain [13]. This means that the convolution is transformed into an element-wise multiplication of two vectors, of size $2L$, with complex entries. Furthermore, combinations of 16 convolutions must be carried out per each sound source. Thus, the operations that are executed by a headphone-based spatial audio application that reproduces M sound sources can be summarized as follows:

1. M FFTs of size $2L$.
2. $16M$ element-wise multiplications between two complex vectors of size $2L$, see (2), and 4 element-wise accumulation of $4M$ vectors of size $2L$.
3. Four inverse FFTs of size $2L$.
4. Four element-wise multiplications between two real vectors of size L, see (3).
5. Two element-wise sums between two real vectors of size L.

Note also that only the input-data buffers $\mathbf{x}_{\text{buff}i}$ must be transformed to the frequency domain since the HRTF filters are already in the database.

Stages 1 and 3 require the use of a FFT library and, to this end, we selected the ad-hoc implementation the *FFTW* library [14] for ARM architectures. Stages 2 and 4 consist of straight-forward element-wise operations and can be implemented as simple loops, to be optimized by the target compiler.

In order to improve this initial implementation, we performed a series of preliminary test to expose the critical path of the algorithm. For this purpose, we executed this implementation using different numbers of sources (M) and found that about 90% or more of the execution time corresponds to stage 2.

5.1. NEON intrinsics for the element-wise multiplications of complex vectors

The FFTW library processes complex data with interleaved real and imaginary parts; see, e.g., Figure 3. In this particular example, the two elements are stored in memory as four 32-bit `float` numbers. In order to exploit data parallelism, we must use SIMD registers and the datatype `float32x4_t` provided by the NEON intrinsics to accommodate them from the source code.

Let us consider three vectors $\mathbf{H}$, $\mathbf{A}$ and $\mathbf{C}$, of the same size L, with their entries disposed as in Figure 3. The element-wise multiplication entailed by stage 2 of the application combines (the real and imaginary parts of) $\mathbf{H}$ and

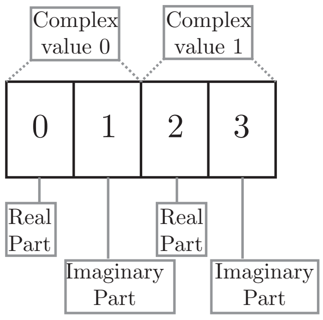

Fig. 3. Conventional storage of a complex vector with two entries.

$\mathbf{A}$ to produce (the real and imaginary parts of) $\mathbf{C}$ as follows:

$$\begin{array}{rclcl} \mathbf{C}[0] & = & \mathbf{H}[0]\cdot\mathbf{A}[0] & - & \mathbf{H}[1]\cdot\mathbf{A}[1] \\ \mathbf{C}[1] & = & \mathbf{H}[0]\cdot\mathbf{A}[1] & + & \mathbf{H}[1]\cdot\mathbf{A}[0] \\ \mathbf{C}[2] & = & \mathbf{H}[2]\cdot\mathbf{A}[2] & - & \mathbf{H}[3]\cdot\mathbf{A}[3] \\ \mathbf{C}[3] & = & \mathbf{H}[2]\cdot\mathbf{A}[3] & + & \mathbf{H}[3]\cdot\mathbf{A}[2] \\ \vdots & = & \vdots & & \vdots \end{array} \tag{4}$$

Unfortunately, when the vectors are stored in memory using this conventional pattern, it is difficult to directly use NEON intrinsics efficiently.

In practice, $\mathbf{H}$ contains the coefficients of the HRTF filters in memory, and its entries are not modified during the filtering. Typically, loads from consecutive (and aligned) memory addresses into vector registers attain a better exploitation of the available bandwidth. Therefore, we can accelerate the loading process of $\mathbf{H}$ into vector registers by a priori reorganizing its contents into two vectors (with some duplicated data): $\mathbf{Hr} = \{\mathbf{H}[0],\mathbf{H}[0],\mathbf{H}[2],\mathbf{H}[2],\ldots\}$ for the real parts and $\mathbf{Hi} = \{\mathbf{H}[1],\mathbf{H}[1],\mathbf{H}[3],\mathbf{H}[3],\ldots\}$ for the imaginary ones.

With this modified layout, an efficient data-parallel implementation of the element-wise complex multiplication can be obtained by means of a sequential application of two NEON intrinsics: `vmulq_f32` which multiplies two vector registers composed of four `floats` element-wise; and `vmlaq_f32`, which multiplies and accumulates the result into a third SIMD register.

Moreover, as vector $\mathbf{A}$ contains the audio samples of a current input-data buffer $\mathbf{x}_{\text{buff}}$ in the frequency domain, we employ two SIMD registers (say `A1` and `A2`) before applying the corresponding multiplications:

```
// Load A[0 ... 3] into A1
A1=vld1q_f32(A);

//Reorganize A1 into A2 as
//A2[0,1,2,3] = A2[1,0,3,2]
A2=vsetq_lane_f32(vgetq_lane_f32(A1,1),A2,0);
A2=vsetq_lane_f32(vgetq_lane_f32(A1,0),A2,1);
A2=vsetq_lane_f32(vgetq_lane_f32(A1,3),A2,2);
A2=vsetq_lane_f32(vgetq_lane_f32(A1,2),A2,3);
```

These registers will be reused 16 times (one per convolution), so the penalty of the rearrangement process is amortized. Besides, three additional SIMD registers (`Hreal`, `Himag` and `C`) are required to carry out the element-wise multiplications, whose values will change during the execution of the 16 convolutions. Concretely, the NEON intrinsic functions that we use for the convolution are:

```
// Load real and imaginary part of H
Hreal=vld1q_f32(Hr);
Himag=vld1q_f32(Hi);

// Compute Equation (4)
C=vmulq_f32(Hreal,A1);
C=vmlaq_f32(C, A2, Himag);
```

where `vgetq_lane_f32`/`vsetq_lane_f322` gets/sets a specific lane from/to a SIMD register, and `vld1q_f32` loads a set of (four) consecutive `floats` from memory to a given SIMD register.

5.2. Performance evaluation

We tested our original and vectorized implementations on a single ARM Cortex-A15 core, running at 1.6 GHz, taking as a reference the time offered by an audio card from a mobile device. This concrete audio device provides 1,024 samples per channel ($L = 1,024$) every 23.22 ms (for a sample frequency $f_s = 44.1$ KHz), which we call buffer time t_{buff}. Assuming that our spatial audio application has to synthesize M sound sources, we define t_{proc} as the processing time, since the M input-data buffers are available till both output-data buffers (for the right and left ear) are totally processed. Thus, the spatial audio application operates in real time provided $t_{\text{proc}} < t_{\text{buff}}$.

The objective of the following experiment is to assess the largest number of sound sources, $M^{\max}$, which can be reproduced in real time on the target architecture using the proposed implementation. In order to determine this factor, we executed the application for an increasing number of sources, and compared the execution time t_{proc} with the threshold t_{buff}. We note that, in case $t_{\text{proc}} > t_{\text{buff}}$, the application does not operate in real time, but can still be considered as an off-line technology.

Figure 4 illustrates the evolution of t_{proc} as a function of the number of sound sources for the original (non-vectorized) implementation and the optimized version with intrinsics. These results demonstrate that, using NEON intrinsics, it is possible to handle up to 60 sound sources simultaneously with a single ARM Cortex-A15 core. Compared with this result, the original implementation could only reproduce up

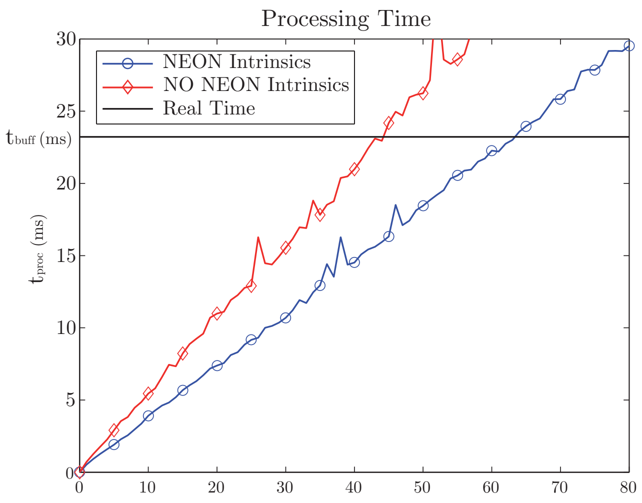

Fig. 4. Execution time of the binaural audio virtualization application implemented with and without NEON intrinsics. The horizontal line at t_{buff} marks the threshold between a real-time application and an off-line technology.

to 40 sound sources in real-time.

6. CONCLUSIONS

Tablets and smart phones are nowadays equipped with low power architectures such as, e.g., the ARMv7 and ARMv8 series, with powerful SIMD units to exploit the ample data-parallelism that characterizes most media and signal processing applications.

In this paper, we have reimplemented an audio processing application that reproduces spatial sound sources in a headphone setting (binaural audio virtualization) and allows the user to interact with their position in real time. In particular, our optimized implementation employs the NEON intrinsic functions to satisfy the high computational demands of this concrete application, fully leveraging the NEON vector floating-point unit present in the ARMv7 architectures. The experimental evaluation on ARM Cortex-A15 core shows that the vectorized code can handle up to 60 sound sources in real time, which is a 33% increase over the implementation that does not utilize the NEON engine.

7. ACKNOWLEDGEMENTS

The researchers from *Universidad Jaime I* were supported by the CICYT project TIN2011-23283 of the *Ministerio de Economía y Competitividad* and FEDER.

REFERENCES

[1] "Arm neon," http://www.arm.com/., (accessed 2015 February 23).

[2] V.R. Algazi and R.O. Duda, "Headphone-based spatial sound," *IEEE Signal Processing Magazine*, vol. 28, no. 1, pp. 33–42, 2011.

[3] Jens Blauert, *Spatial Hearing - Revised Edition: The Psychophysics of Human Sound Localization*, The MIT Press, 1996.

[4] G. Mitra, B. Johnston, A.P. Rendell, E. McCreath, and Jun Zhou, "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)*, May 2013, pp. 1107–1116.

[5] E. Welch, D. Patru, E. Saber, and K. Bengtson, "A study of the use of simd instructions for two image processing algorithms," in *Western New York Image Processing Workshop (WNYIPW)*, Nov 2012, pp. 21–24.

[6] Ronggang Wang, Jie Wan, Wenmin Wang, Zhenyu Wang, Shengfu Dong, and Wen Gao, "High definition ieee avs decoder on arm neon platform," in *20th IEEE International Conference on Image Processing (ICIP)*, Sept 2013, pp. 1524–1527.

[7] S.B. Holgersson, "Optimising iir filters using arm neon," M.S. thesis, University of Danemark, 2012.

[8] Yuvi Kahana and Philip A. Nelson, "Numerical modelling of the spatial acoustic response of the human pinna," *Journal of Sound and Vibration*, vol. 292, no. 1-2, pp. 148 – 178, 2006.

[9] "Listen HRTF database," *online at: http://recherche.ircam.fr/equipes/salles/listen/index.html*.

[10] Akihiro Kudo, Haruhide Hokari, and Shoji Shimada, "A study on switching of the transfer functions focusing on sound quality," *Acoustical Science and Technology*, vol. 26, no. 3, pp. 267–278, 2005.

[11] J. A. Belloch, M. Ferrer, A. Gonzalez, F.J. Martinez-Zaldivar, and A. M. Vidal, "Headphone-based virtual spatialization of sound with a GPU accelerator," *J. Audio Eng. Soc*, vol. 61, no. 7/8, pp. 546–561, 2013.

[12] "Armv7 neon data types," https://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/ARM-NEON-Intrinsics.html., (accessed 2015 February 26).

[13] A. V. Oppenheim, A. S. Willsky, and S. Hamid, "Signals and systems," Processing series. Prentice Hall, 2nd edition, 1997.

[14] "Fast Fourier Transform West," http://www.fftw.org/, (accessed 2015 January 11).