# Optimized Fundamental Signal Processing Operations for Energy Minimization on Heterogeneous Mobile Devices

*Jose A. Belloch, José M. Badía, Francisco D. Igual, Alberto Gonzalez, and Enrique S. Quintana-Ortí*

## Abstract

Numerous signal processing applications are emerging on both mobile and high-performance computing systems. These applications are subject to responsiveness constraints for user interactivity and, at the same time, must be optimized for energy efficiency. The increasingly heterogeneous power-versus-performance profile of modern hardware introduces new opportunities for energy savings as well as challenges. In this line, recent Systems-On-Chip (SoC) composed of low-power multicore processors, combined with a small graphics accelerator (or GPU), yield a notable increment of the computational capacity while partially retaining the appealing low power consumption of embedded systems. This paper analyzes the potential of these new hardware systems to accelerate applications that involve a large number of floating-point arithmetic operations mainly in the form of convolutions. To assess the performance, a headphone-based spatial audio application for mobile devices based on a Samsung Exynos 5422 SoC has been developed. We discuss different implementations and analyze the trade-offs between performance and energy efficiency for different scenarios and configurations. Our experimental results reveal that we can extend the battery lifetime of a device featuring such an architecture by a 238% by properly configuring and leveraging the computational resources.

**Keywords:** Audio systems, signal synthesis, parallel architectures, parallel processing, heterogeneous (hybrid) systems, performance and energy efficiency.

## 1 Introduction

Low-power (embedded) processors play an important role for a myriad of signal processing applications, such as communications [1–3], video coding [4–6], image processing [7], visual detection [8], speech recognition [9], and audio processing [10], among others. In the era of smart phones and tablets, these energy-efficient architectures have increased significantly their computational capacity and are nowadays utilized in a large volume of multimedia (video and audio processing).

The Exynos 5422 system-on-chip (SoC) is an implementation of the ARM big.LITTLE architecture [11] present in multiple mobile devices and development boards, such as the Samsung Galaxy S5 (SM-G900H) and the ODROID boards XU3/XU3-Lite/XU4, respectively. This SoC combines a quad-core ARM Cortex-A15 cluster with a quad-core ARM Cortex-A7, and each core integrates a 128-bit NEON SIMD (single-instruction, multiple-data) engine. In addition, the Exynos 5422 includes a Mali-T628 MP6 GPU with six shader cores.

This work analyzes the potential of these new hardware systems to accelerate signal processing applications that involve a moderate number of convolutions. Specifically, we focus on a spatial audio system based on headphones that requires multiple convolutions for synthesizing a virtual sound source [12]. The target application, for example, allows the listener to virtually change the positions of the sound sources (guitar, drums, voice, etc. ) of a song that is reproduced throughout the headphones of a mobile device [13,14]. This effect is achieved by processing the sound samples through a collection of special filters that shape the sound with spatial information (convolution). In the frequency domain, these filters are known as *Head-Related Transfer Functions* (HRTFs), where the response of a HRTF describes how a sound wave is affected by certain properties of the individual's body (e.g. pinna, head, shoulders, neck and torso) before the sound reaches the listener's eardrum [15]. The cost of the computational problem that underlies a headphone-based application grows linearly with the number of independent sound sources that have to be handled (i.e., reproduced and moved) in real time.

A range of low-power ARM processors have been previously leveraged in audio and video signal processing scenarios [16]. For example, some image processing algorithms are accelerated on this type of architectures in [17], and an image processing application is optimized using NEON intrinsics in [18]. ARM-based platforms have also been used to accelerate the Audio Video coding Standard (AVS) in [19–21], and various types of filters have been implemented for audio filtering using NEON extensions in [22]. On the other hand, different audio filter structures have been accelerated with NEON intrinsic functions in [23].

However, NEON intrinsics functions have not been leveraged yet to compute convolutions in the frequency domain on ARM processors, which requires the application of Fourier transforms as well as complex arithmetic. An initial study of the HSA algorithm, using a single ARM Cortex-A15 core only, was previously presented by the authors in [24].

In this paper we introduce an efficient code for a headphone-based spatial audio application (HSA) for the Exynos 5422 SoC that presents high versatility, since our implementation can be run on the ARM Cortex-A15, the ARM Cortex-A7, or the Mali-T628 MP6 GPU. The HSA could be executed on either the ARM Cortex-A7 or the Mali-T628 MP6 GPU in case of running the HSA application simultaneously with another application. This could be the case, for example, of an image-based application such as a video game, which could require access to the ARM Cortex-A15 for compute-intensive tasks. Given the growing importance of mobile appliances, and the direct relation between energy consumption and battery lifetime in these devices, in this work we discuss different implementations and analyze the trade-offs between performance and energy efficiency for different distributions of the computational load on the Exynos 5422 SoC and regulating the frequency of its components.

The analysis in our work gains in relevance because we target a low-cost platform that is currently present in multiple mobile devices. Thus, we expect that our results partially carry over to other platforms that are designed for the same purpose. Furthermore, HSA comprises all fundamental operations of a signal processing application and, indeed, HSA is a trending application that could be embedded into more complex applications based on virtual reality or video-games [25]. Hence, running this application efficiently, from the points of view of performance and energy consumption, in common mobile devices is an interesting contribution to analyze the growing possibilities of mobile devices.

The main specific contributions of this paper can be summarized as follows:

- Explore the versatility of low-power platforms to deal efficiently with compute-intensive applications.
- Implement several efficient and portable algorithms to solve a massive data-parallel audio processing method on a range of heterogeneous platforms that integrate multiple CPUs and/or GPUs.
- Exploit and combine the heterogeneous parallel components of a low-power platform including its multicore CPU and GPU by means of standard programming tools such as OpenMP and OpenCL, and also leveraging the specific SIMD capabilities of the ARMv7 architecture.
- Exploit the capability of the architecture to regulate the frequency of some of its components or even disable them in order to reduce the power consumption.
- Analyze different configurations in terms of algorithm, platform components, and frequencies to optimize the application in different scenarios. These scenarios include the reduction of the total time, the optimization of the number of audio sources processed per Watt, or the increase of the battery lifetime.

The rest of the paper is structured as follows: In Section 2 we introduce the HSA application. Next, in Section 3, we present several features of the Exynos 5422, including the SIMD capabilities of the NEON vector floating-point units for programming ARM Cortex CPUs, and the OpenCL interface for programming its GPU. Sections 4 and 5 explain in detail the implementations on the ARM Cortex and the GPU, respectively. These two sections illustrate performance in terms of maximum number of sound sources that can be rendered in real time; provide a detailed analysis of the power dissipation; and analyze the energy efficiency of different hardware configurations. Furthermore, the results in Section 5 offer a guide to select the configuration depending on the purpose of the application. For example, the goal may be minimizing energy consumption (to extend battery lifetime) while meeting a certain quality-of-service in terms of number of moving sound sources. Finally, Section 6 presents some concluding remarks.

## 2 Spatial audio on Headphones

A variety of 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 to the audio wave from specific positions in the space. For this purpose, in headphone-based systems, one *head-related impulse response* (HRIR) filter per ear specifies each virtual position in the time domain. There are multiple public samples of HRIRs for the time domain as well as *Head-Related Transfer Functions* (HRTF) for the frequency domain. In our case, we leverage the HRIR measures from [26]. 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 metrics 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. The HRTF filters are obtained off-line, in our case, via the Fourier transform of the respective HRIR data. Although our algorithm operates in the frequency domain, for simplicity we adhere to the time domain in the following description.

Let us employ $\mathbf{x}_{\mathcal{B}_i}$ to denote an input-data buffer $\mathcal{B}$ consisting of $L$ audio samples, from a sound source $x_i$, with $i \in [0, M-1]$ and $M$ representing the number of sources. In addition, assume that the HRIRs corresponding to position $(\theta, \phi)$ in the time domain are given by $\mathbf{h}_\mathrm{r}(\theta, \phi)$ and $\mathbf{h}_\mathrm{l}(\theta, \phi)$, for the right and left ear, respectively. The output-data buffer, for both the right and left ears, $\mathbf{y}_\mathcal{B}$, is then given in the time domain by

$$\mathbf{y}_\mathcal{B} = \sum_{i=0}^{M-1} \mathbf{y}_{\mathcal{B}_i}(\theta_i, \phi_i) = \sum_{i=0}^{M-1} (\mathbf{h}(\theta_i, \phi_i) * \mathbf{x}_{\mathcal{B}i}), \tag{1}$$

where $*$ stands for the convolution operator.

In practice, the number of filters in the database is limited, constraining the virtual positions that can be rendered in real time. Consider, for example, a position $(\theta_S, \phi_S)$ that is not in the database, surrounded by four positions in the database, say $\{(\theta_1, \phi_1), (\theta_1, \phi_2), (\theta_2, \phi_1), (\theta_2, \phi_2)\}$. To tackle this scenario, our solution employs four filters to synthesize this position via linear interpolation, yielding

$$\begin{aligned} \mathbf{y}_{\mathcal{B}_i}(\theta_S, \phi_S) \;&=\; \mathbf{x}_{\mathcal{B}i} * \big(w_D \cdot w_B \cdot \mathbf{h}(\theta_1, \phi_1) \\ &+\; w_D \cdot w_A \cdot \mathbf{h}(\theta_2, \phi_1) \\ &+\; w_C \cdot w_B \cdot \mathbf{h}(\theta_1, \phi_2) \\ &+\; w_C \cdot w_A \cdot \mathbf{h}(\theta_2, \phi_2)\big), \end{aligned} \tag{2}$$

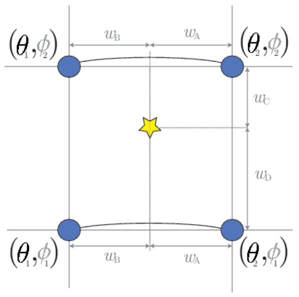


Fig. 1: Relations between the interpolation weights and the distances on the data base plane. The star represents the position $(\phi_S, \theta_S)$ to be synthesized in the elevation and azimuth planes.

where $\{w_A, w_B, w_C, w_D\}$ represent the interpolation weight; see Figure 1. However this enhancement comes at the cost of a notable increase in the computational complexity.

A second challenge occurs when the sound source is moving, which in practice requires the careful application of a sequence of new (HRIR) filters in a fade-in fade-out strategy. In particular, if the commutation between HRIRs is not properly implemented, this may result in undesirable audio artifacts [27]. To deal with this, one additional filtering is necessary for the virtualization of source movement, which is carried out by smoothly varying the virtual positions of the source over time. Concretely, assume the sound source $x_i$ moves from position A: $(\theta_{SA}, \phi_{SA})$ to position B: $(\theta_{SB}, \phi_{SB})$. We then implement the fading as a gradual increase in the sound filtered by position B while the sound filtered by position A attenuates in the same proportion. To this end, the current output-data buffer $\mathbf{y}_{\mathcal{B}_i}$ is computed for both positions, and the result of both computations is then multiplied element-wise by two fading vectors, say $\mathbf{f}$ and $\mathbf{g}$. This kind of multiplication is also know as a Hadamard product. Finally, the output-data buffer $\mathbf{y}_{\mathcal{B}_i}$ is obtained by adding the results from the previous multiplications element-wise:

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

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

## 3 Exploring the Exynos 5422

The Exynos 5422 SoC contains a quad-core ARM Cortex-A15 cluster, a quad-core ARM Cortex-A7, and a Mali-T628 MP6 GPU, integrated with the memory system hierarchy illustrated in Figure 2. The next subsections review the

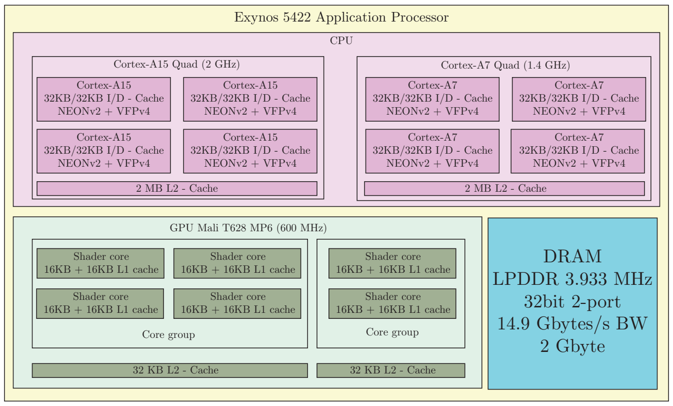


Fig. 2: Diagram of the processor Exynos 5422 included in the Odroid XU3 experimental platform.

SIMD capabilities of the NEON floating-point units ARM Cortex CPUs, and the OpenCL standard for programming its GPU.

One of the most interesting features of this platform is the possibility of adjusting its energy consumption by reducing the frequency of the CPU cores or the GPU, or even by disabling some of these components. In our experiments, we measured the power dissipation of the Exynos 5422 using the `pmlib` framework [28] to collect the instantaneous power readings from the internal energy-monitoring sensors [29]. The ODROID XU3 board contains four real time current sensors that can be sampled to obtain the power consumption of four separate power domains: Cortex-A15 cores, Cortex-A7 cores, DRAM and Mali GPU. In the experiments performed with the Cortex-A7 cores, we disabled the Cortex-A15 cores to avoid the effect of their non-negligible power dissipation even when they are not being used. We could have obtained very similar results simply by not adding the effect of the Cortex-A15 power domain, but our goal is to measure the total power consumption of the platform and to exploit the possibility of disabling some of its components.

## 3.1 The NEON SIMD engine

The ARMv7 architecture implements the SIMD technology by means of the NEON vector floating-point unit. This component supports a variety of SIMD floating-point instructions to simultaneously apply the same operation on a collection of packed elements of the same type and size, including arithmetic and logical operations and data movements. In order to support this functionality, the ARMv7 architecture comprises 32 64-bits SIMD registers, (also viewed as 16 128-bits registers,) which can be leveraged to operate with two 64-bit floating-point numbers, four 32-bit floating-point/integer numbers, or eight 16-bit integers. Each item stored in an SIMD register is usually referred to as a *lane*. The ARMv7 ISA includes SIMD instructions for arithmetic and logical operations (addition, multiplication, fused multiply-add, multiply-subtract, subtraction, comparison, etc.) and data movements (e.g., load/store lanes from memory/SIMD register into an SIMD register/memory).

The SIMD technology is realized in the ARMv7 architecture as part of the NEON vector floating-point unit. The vector register set in the NEON engine and the general-purpose register set in the ARM core are independent. Moreover, the ARMv7 SIMD operations can be cast in terms of a few intrinsic functions that are directly translated by the compiler into NEON instructions, offering a simpler programming interface.

The ARMv7 NEON intrinsics implement all the functionality of the NEON instruction set, including the definition of vector data types [30] to place data into vector registers. These types take the form *"type" "size" x "number of lanes" _t*, with the following list of relevant examples `int8x8_t`, `int8x16_t`, `int16x4_t`, `int32x4_t`, `int64x2_t`, `float32_t`, `float32x2_t`, and `float32x4_t`.

## 3.2 OpenCL

OpenCL (Open Computing Language) [31] is an open royalty-free standard for general-purpose parallel programming across CPUs, GPUs and other processors. It is supported by a wide range or different architectures, easing the development of portable parallel software for heterogeneous platforms.

OpenCL defines a hierarchy of models to represent the platform, memory, execution and programming of the architecture as well as to provide an abstract view of the hardware that allows the development of portable software. At the same time, programmers can access multiple features of the platform at runtime to adapt the code to specific architectures and improve its performance.

An OpenCL platform comprises a set of possibly heterogeneous computing devices (GPUs, CPUs, etc.). Each device consists of different compute units (CU) containing multiple processing elements (PE). The compute units usually correspond to the cores of the CPU or GPU, though the notion of core is delicate to define across the multiple types of devices supported by OpenCL.

In the OpenCL programming model, data-parallelism is exploited by dividing the computation into multiple *work-items* that can run the same code in parallel on different processing elements over different data. The code executed by each *work-item* is included in kernels that are submitted by the host to the devices.

## 4 Implementation and Tuning of the HSA Algorithm on the EXYNOS 5422

In our implementation, the convolution operations in the HSA algorithm are carried out using the overlap-save technique with a 50% overlap in the frequency domain [32, 33]. This means that we are processing the $L$ audio samples that are in the current input-data buffer together with the $L$ audio samples from the previous input-data buffer. Each convolution in the time domain turns into an element-wise multiplication of two vectors, of size $2L$, with complex entries. Thus, the operations that are executed by the HSA algorithm, in order to reproduce $M$ sound sources, can be summarized in the following stages:

1. $M$ Fast Fourier Transforms (FFTs) of size $2L$.
2. $16M$ element-wise vector sums ($4M$ for left ear and position $A$, $4M$ for left ear and position $B$, $4M$ for right ear and position $A$, and $4M$ for right ear and position $B$), and $4M$ element-wise vector multiplications. All vectors are composed of complex entries with a size $2L$; see (2).
3. Four inverse FFTs of size $2L$.
4. Four element-wise multiplications of two real vectors of size $L$; see (3).
5. Two element-wise additions of two real vectors of size $2L$.
6. These two real vectors contain the audio samples that are reproduced by the right and the left headphone, respectively.

Note also that only the input-data buffers $\mathbf{x}_{\mathcal{B}_i}$ must be transformed to the frequency domain in a real-time scenario, since the HRTF filters are obtained after the FFT of HRIR in off-line mode.

Stages 1 and 3 require the computation of FFTs, which are performed in our code via the implementation available in the *FFTW* library for ARM architectures [34], and the OpenCL library implementation of discrete Fast Fourier Transforms clFFT [35] for GPUs. For the library [34], we use the half-complex format that returns a vector of the non-redundant half of the complex output for a one dimension real-input DFT of size $2L$, stored as a sequence of $2L$ real numbers (`floating-point elements`) in the format ($r$ for real component and $i$ for imaginary component of a complex number):

$$r_0, r_1, r_2, ..., r_{2L/2}, i_{(2L+1)/2-1}, ..., i_2, i_1$$

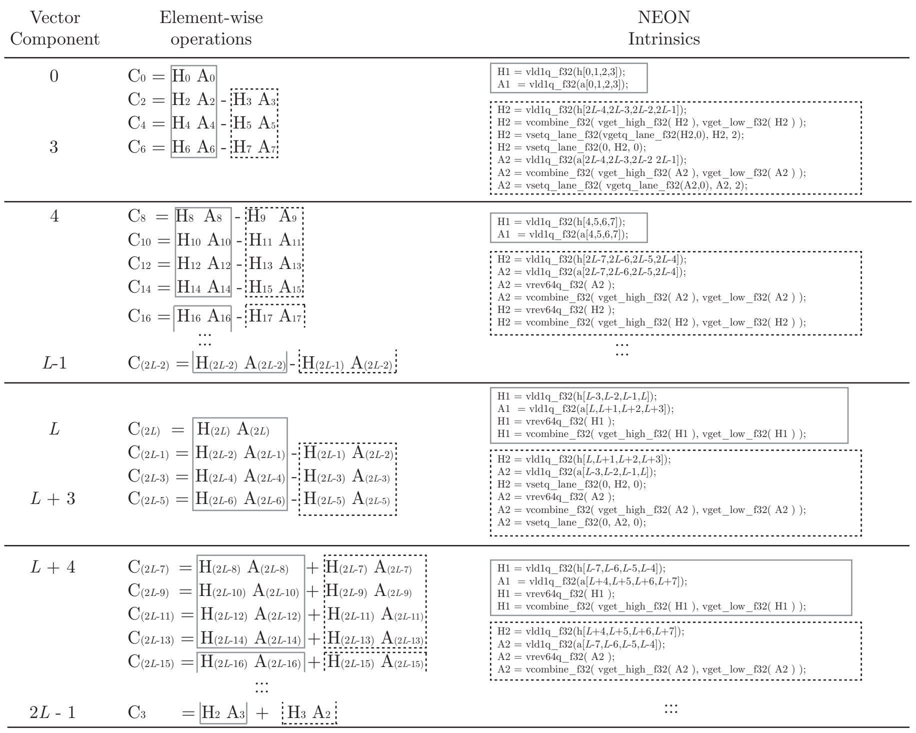
In contrast to these sophisticated codes, stages 2, 4 and 5 consist of element-wise operations that can be implemented as plain loops, to be optimized by the target compiler.

The following subsections describe the implementation plus optimization on each type of computational resource available in the Exynos 5422 SoC: ARMv7 processors and ARM Mali GPU.

## 4.1 Implementation on ARMv7 processors

In order to reveal the critical path and/or computational bottlenecks of the algorithm, we first implemented a sequential code for the operations described in the previous section, using different number of sound sources ($M$). This initial analysis revealed that 90% or more of the execution time corresponds to stage 2 (the element-wise sums and multiplications), that we next target in our optimization effort.

In order to exploit data-parallelism, we use the SIMD registers and the data type `float32x4_t` provided by the NEON intrinsics. Figure 3 shows the operations that are carried out using two input vectors, **H** and **A**, in order



| Vector Component | Element-wise operations | NEON Intrinsics |
|---|---|---|
| 0<br><br><br>3 | $C_0 = \boxed{H_0\ A_0}$<br>$C_2 = \boxed{H_2\ A_2} - \boxed{H_3\ A_3}$<br>$C_4 = \boxed{H_4\ A_4} - \boxed{H_5\ A_5}$<br>$C_6 = \boxed{H_6\ A_6} - \boxed{H_7\ A_7}$ | H1 = vld1q_f32(h[0,1,2,3]);<br>A1 = vld1q_f32(a[0,1,2,3]);<br><br>H2 = vld1q_f32(h[2*L*-4,2*L*-3,2*L*-2,2*L*-1]);<br>H2 = vcombine_f32( vget_high_f32( H2 ), vget_low_f32( H2 ) );<br>H2 = vsetq_lane_f32(vgetq_lane_f32(H2,0), H2, 2);<br>H2 = vsetq_lane_f32(0, H2, 0);<br>A2 = vld1q_f32(a[2*L*-4,2*L*-3,2*L*-2 2*L*-1]);<br>A2 = vcombine_f32( vget_high_f32( A2 ), vget_low_f32( A2 ) );<br>A2 = vsetq_lane_f32( vgetq_lane_f32(A2,0), A2, 2); |
| 4<br><br><br><br><br><br>*L*-1 | $C_8 = \boxed{H_8\ A_8} - \boxed{H_9\ A_9}$<br>$C_{10} = \boxed{H_{10}\ A_{10}} - \boxed{H_{11}\ A_{11}}$<br>$C_{12} = \boxed{H_{12}\ A_{12}} - \boxed{H_{13}\ A_{13}}$<br>$C_{14} = \boxed{H_{14}\ A_{14}} - \boxed{H_{15}\ A_{15}}$<br>$C_{16} = \boxed{H_{16}\ A_{16}} - \boxed{H_{17}\ A_{17}}$<br>$\vdots$<br>$C_{(2L-2)} = \boxed{H_{(2L-2)}\ A_{(2L-2)}} - \boxed{H_{(2L-1)}\ A_{(2L-2)}}$ | H1 = vld1q_f32(h[4,5,6,7]);<br>A1 = vld1q_f32(a[4,5,6,7]);<br><br>H2 = vld1q_f32(h[2*L*-7,2*L*-6,2*L*-5,2*L*-4]);<br>A2 = vld1q_f32(a[2*L*-7,2*L*-6,2*L*-5,2*L*-4]);<br>A2 = vrev64q_f32( A2 );<br>A2 = vcombine_f32( vget_high_f32( A2 ), vget_low_f32( A2 ) );<br>H2 = vrev64q_f32( H2 );<br>H2 = vcombine_f32( vget_high_f32( H2 ), vget_low_f32( H2 ) );<br>$\vdots$ |
| *L*<br><br><br>*L* + 3 | $C_{(2L)} = \boxed{H_{(2L)}\ A_{(2L)}}$<br>$C_{(2L-1)} = \boxed{H_{(2L-2)}\ A_{(2L-1)}} - \boxed{H_{(2L-1)}\ A_{(2L-2)}}$<br>$C_{(2L-3)} = \boxed{H_{(2L-4)}\ A_{(2L-4)}} - \boxed{H_{(2L-3)}\ A_{(2L-3)}}$<br>$C_{(2L-5)} = \boxed{H_{(2L-6)}\ A_{(2L-6)}} - \boxed{H_{(2L-5)}\ A_{(2L-5)}}$ | H1 = vld1q_f32(h[*L*-3,*L*-2,*L*-1,*L*]);<br>A1 = vld1q_f32(a[*L*,*L*+1,*L*+2,*L*+3]);<br>H1 = vrev64q_f32( H1 );<br>H1 = vcombine_f32( vget_high_f32( H1 ), vget_low_f32( H1 ) );<br><br>H2 = vld1q_f32(h[*L*,*L*+1,*L*+2,*L*+3]);<br>A2 = vld1q_f32(a[*L*-3,*L*-2,*L*-1,*L*]);<br>H2 = vsetq_lane_f32(0, H2, 0);<br>A2 = vrev64q_f32( A2 );<br>A2 = vcombine_f32( vget_high_f32( A2 ), vget_low_f32( A2 ) );<br>A2 = vsetq_lane_f32(0, A2, 0); |
| *L* + 4<br><br><br><br><br><br>2*L* - 1 | $C_{(2L-7)} = \boxed{H_{(2L-8)}\ A_{(2L-8)}} + \boxed{H_{(2L-7)}\ A_{(2L-7)}}$<br>$C_{(2L-9)} = \boxed{H_{(2L-10)}\ A_{(2L-10)}} + \boxed{H_{(2L-9)}\ A_{(2L-9)}}$<br>$C_{(2L-11)} = \boxed{H_{(2L-12)}\ A_{(2L-12)}} + \boxed{H_{(2L-11)}\ A_{(2L-11)}}$<br>$C_{(2L-13)} = \boxed{H_{(2L-14)}\ A_{(2L-14)}} + \boxed{H_{(2L-13)}\ A_{(2L-13)}}$<br>$C_{(2L-15)} = \boxed{H_{(2L-16)}\ A_{(2L-16)}} + \boxed{H_{(2L-15)}\ A_{(2L-15)}}$<br>$\vdots$<br>$C_3 = \boxed{H_2\ A_3} + \boxed{H_3\ A_2}$ | H1 = vld1q_f32(h[*L*-7,*L*-6,*L*-5,*L*-4]);<br>A1 = vld1q_f32(a[*L*+4,*L*+5,*L*+6,*L*+7]);<br>H1 = vrev64q_f32( H1 );<br>H1 = vcombine_f32( vget_high_f32( H1 ), vget_low_f32( H1 ) );<br><br>H2 = vld1q_f32(h[*L*+4,*L*+5,*L*+6,*L*+7]);<br>A2 = vld1q_f32(a[*L*-7,*L*-6,*L*-5,*L*-4]);<br>A2 = vrev64q_f32( A2 );<br>A2 = vcombine_f32( vget_high_f32( A2 ), vget_low_f32( A2 ) );<br>$\vdots$ |

Fig. 3: Operations that must be carried out between two vectors, **H** and **A**, in order to compute vector **C** in the half-complex format.

to compute vector **C** in the half-complex format (all vectors have a size of $2L$). The first column of this figure indicates the vector components and the second column shows the computations. The subscripts of the resulting vector **C** correspond to the half-complex format and that the elements of **H** and **A** are stored in the same way. Except for the computation of two vector elements of **C**, all others are composed of a sum of two addends which come from a multiplication of two different elements of vectors **H** and **A**.

By properly exploiting the SIMD registers, we can compute four components of vector **C** at a time. Loading the components of vectors **H** and **A** in both addends requires the use of the NEON shuffle instructions. The third column in Figure 3 shows the NEON instructions that store four values of **A** and **H** in the SIMD registers, in order to be element-wise multiplied. The grey line and the black dashed-line rectangles link the elements of each addend with their corresponding load and element-arranging NEON instructions.

Because of the layout of the half-complex format, we split the computation of all vector components into four blocks as shown in Figure 3. It is important to note that blocks 1 and 3 compute only four elements at a time, while blocks

2 and 4 compute $L-4$ elements in steps of four elements.

As depicted in stage 2 at the beginning of this section, prior to the element-wise multiplication between vectors $\mathbf{A}$ and $\mathbf{H}$, it is necessary to carry out 16 element-wise vector sums with their corresponding interpolation weights, according to (2). To this end, we execute the following NEON instructions:

```
// For the first weighting.
   Ht = vmulq_n_f32(Hi, w);
// For weighting and accumulation
   Ht =  vmlaq_n_f32 (Ht, Hi, w);
```

where `Hi` represents a SIMD register that contains four elements of any of the HRTF filters of (2) which are weighted and accumulated in the SIMD register `Ht`.

Once the SIMD registers contain the proper values, we use the following NEON intrinsics: `vmulq_f32` to multiply two vector registers composed of four `float`s element-wise; `vmlsq_f32` to multiply and subtract the result into a third SIMD register (vector components from 0 to $L-1$, see Figure 3); and `vmlaq_f32` to multiply and accumulate the result into a third SIMD register (vector components from L to $2L-1$, see Figure 3).

## 4.2 Multi-threaded parallelization

After the data reorganization to accelerate the execution with NEON intrinsics, the operations for stage 2 are structured as follows:

```
   // Execution each 4 samples
{1} for( l = 0; l < 2*L; l = l + 4 )
      // For each sound source
  {2} for( m = 0; m < M; m = m + 1 )
          //For position A and B
      {3} for( n = 0; m < 2; n = n + 1 )
            //For Left and Right Ears.
        {4} for( z = 0; z < 2; z = z + 1 )
              //For the four filters.
          {5} for( j = 0; j < 4; j = j + 1 )
            // Element-wise vector sums
            //  and multiplications
```

Here loop {1} is unrolled in 4 blocks (see Figure 3); and loops {3} and {4} are also unrolled since stage 3 requires four vectors of size $2L$ as input: two vectors (Right and Left ears) for each position (Position A and B).

In order to exploit the hardware concurrency of the quad-core clusters, we leverage OpenMP to parallelize the second and the fourth block of loop {1} above, setting the following two lines before it:

```
omp_set_num_threads(nth);
#pragma omp parallel for
private (m,n,z,j)
```

Here `nth` specifies the number of threads. We have tested different scheduling strategies provided by OpenMP obtaining the best results with the default static scheduling. In addition, the invocations to the FFTs in the FFTW library are computed in parallel by initializing the library with the following commands:

```
fftwf_init_threads();
fftwf_plan_with_nthreads(nth);
```

## 4.3 Implementation on the ARM Mali GPU

The Mali-T628-MP6 GPU includes 6 identical shader cores and, when programmed with OpenCL, it exposes two heterogeneous devices, one consisting of four compute units (shader cores) and the other featuring the remaining two. We have implemented two different parallel versions of the HSA algorithm using this GPU. The first approach leverages only the device composed of four compute units and the second approach exploits both of them. Figure 4 summarizes the tasks that carry out all of the *work-items* for each launched kernels.

### 4.3.1 First Approach

In order to carry out all stages of the HSA algorithm, we implemented four different OpenCL kernels. Algorithm 1 summarizes the main steps carried out and the input and output data of each kernel using one device. Note that we make use of the clFFT library to carry out the corresponding the FFT (forward transformation) and the inverse FFT (backward transformation). Steps 3-8 of Algorithm 1 are carried out in the GPU. We next describe in more detail the kernels that we have implemented and used in Algorithm 1.

**Algorithm 1** GPU-based implementation using one device

```
1: function HSA($\mathbf{x}$, $\mathbf{H}$, $M$, $w_A$, $w_B$, $w_C$, $w_D$, $\mathbf{f}$, $\mathbf{g}$)
2:     Transfer CPU $\rightarrow$ GPU: $\mathbf{x}, w_A, w_B, w_C, w_D$
3:     $\mathbf{X} \leftarrow$ clFFT($\mathbf{x}$, ‘’forward”)
4:     $\mathbf{RH}$ = kBuildFilters($\mathbf{H}$, $w_A$, $w_B$, $w_C$, $w_D$, $L$)
5:     $\mathbf{Y}$ = kMult($\mathbf{X}, \mathbf{RH}$, $L$)
6:     $\mathbf{RY}$ = kRed($\mathbf{Y}$, $M$, $L$)
7:     $\mathbf{y}$ = clFFT($\mathbf{RY}$, ‘’backward”);
8:     $\mathbf{fy}$ = kFade($\mathbf{y}$, $\mathbf{f}$, $\mathbf{g}$, $L$)
9:     Transfer CPU $\leftarrow$ GPU: $\mathbf{fy}$
10:    return $\mathbf{fy}$
11: end function
```

- `kBuildFilters`. This kernel weights and combines four selected filters in $\mathbf{H}$ (the four filters $\mathbf{h}$ in (2) in the frequency domain) per channel into one. Each filter corresponds to one of the four positions shown in Figure 1.

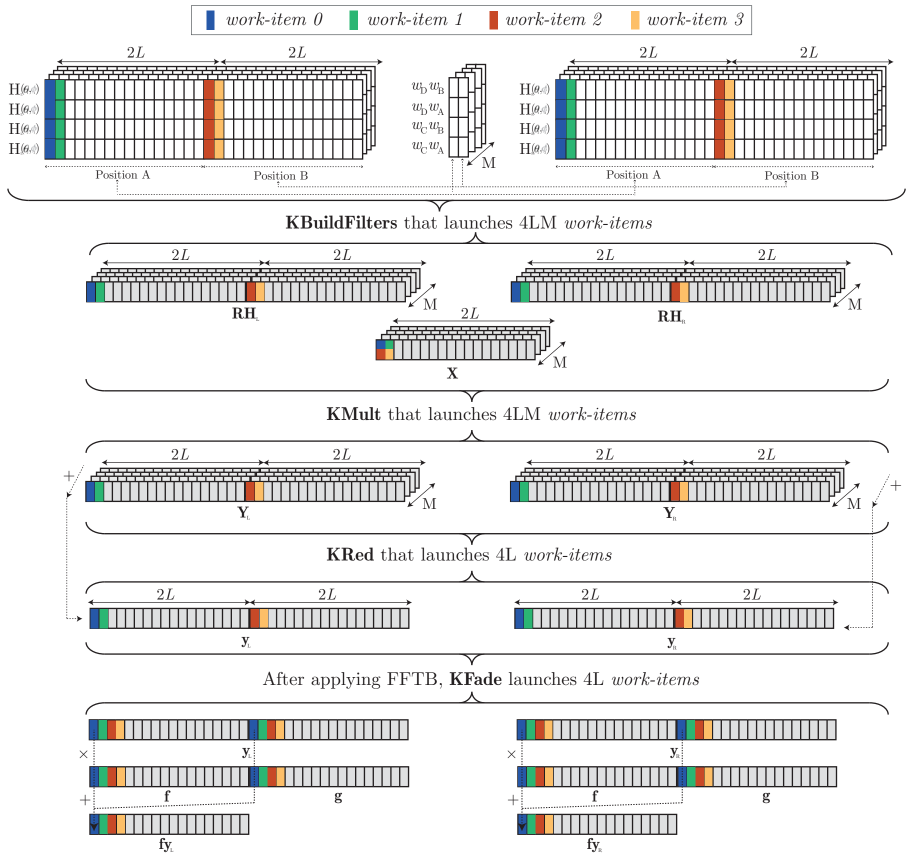

work-item 0
work-item 1
work-item 2
work-item 3
2L
H
Position A
Position B
M
KBuildFilters that launches 4LM work-items
RH
X
KMult that launches 4LM work-items
Y
KRed that launches 4L work-items
y
After applying FFTB, KFade launches 4L work-items
f
g
fy

Fig. 4: Tasks that carry out all of the *work-items* in each one of the launched kernels.

To this end we use a 2D workspace of $M \times 4L$ *work-items*. Each *work-item* weights and combines four complex elements per channel into one. Algorithm 2 shows the main steps of this kernel.

- `kMult`. This kernel also uses a 2D workspace of $M \times 4L$ *work-items*. Each *work-item* performs 2 products, one per channel, between a complex value of $\mathbf{X}$ (Fourier transform of all $\mathbf{x}_{\mathcal{B}i}$ in (2)) and a complex value of $\mathbf{RH}$. The output is composed of $4M$ complex vectors of size $2L$ denoted as $\mathbf{Y}$. Algorithm 3 shows the main steps of this kernel.

We have implemented and compared several versions of our kernels using different computational granularities and access patterns to the data. For example, for kernel `kMult` we could have performed each complex product corresponding to one of the channels in parallel. We have tested several combinations of the products grouping sources, channels and positions on

**Algorithm 2** Combination of the HRTF filters

```
1: function kBuildFilters($\mathbf{H}$, $w_A$, $w_B$, $w_C$, $w_D$, $L$)
2:     $oh$ = offset in $\mathbf{RH}_L$ and $\mathbf{RH}_R$
3:     // $w_A$, $w_B$, $w_C$, $w_D$ depend on source and position
4:     for f ← 1 to 4 do // for each filter
5:         Compute $w_f$ from $w_A, w_B, w_C, w_D$
6:         $oa$ = offset in $\mathbf{H}$
7:         $\mathbf{RH}_L[oh]$ += $w_f * \mathbf{H}[oa]$
8:         $\mathbf{RH}_R[oh]$ += $w_f * \mathbf{H}[oa]$
9:     end for
10:    return RH
11: end function
```

**Algorithm 3** Application of the filters

```
1: function kMult($\mathbf{X}$, $\mathbf{RH}$, $L$)
2:     $ox$ = offset in $\mathbf{X}$
3:     $o$ = offset in $\mathbf{H}_L$, $\mathbf{H}_R$, $\mathbf{Y}_L$ and $\mathbf{Y}_R$
4:     $\mathbf{Y}_L[o] = \mathbf{RH}_L[o] * \mathbf{X}[ox]$
5:     $\mathbf{Y}_R[o] = \mathbf{RH}_R[o] * \mathbf{X}[ox]$
6:     return Y
7: end function
```

each *work-item*. The best results were obtained with the version of the kernel shown in Algorithm 3. It is also important to implement a coalesced access to the memory so that the *work-items* on each *work-group* access adjacent elements in memory. We have also tested different *work-group* sizes with the best results for all kernels obtained using the maximum size allowed by the ODROID platform, that is, 256 *work-items* per *work-group*. For the 2D workspaces the best results where obtained with *work-groups* of size $1 \times 256$

- `kRed`. This kernel reduces the $4M$ vectors $\mathbf{Y}$ to 4 vectors $\mathbf{y}$ (right and left, and each one, twice for the movement virtualization). To this end, we use a 1D workspace and launch $4L$ *work-items*. Each *work-item* reduces $2M$ complex elements (a single element per source of the right and left channels of one of the positions A and B), to two elements, one per channel: see Fig. (4). Algorithm 4 shows the main steps of this kernel.

  We have also implemented a version of the kernel that deals with only one of the channels obtaining similar performances. We could have also parallelized the computations of each reduction, but then we would have needed to coordinate the execution of several *work-items* and combine their results to get each element of $\mathbf{y}$.

- `kFade`. This kernel multiplies element-wise the four $\mathbf{y}$ vectors with the fading vectors $\mathbf{f}$ and $\mathbf{g}$, and next reduces them in order to obtain the two

**Algorithm 4** Reduction of the filtered samples

```
1: function KRED($\mathbf{Y}$, $M$, $L$)
2:     $o$ = offset in $\mathbf{Y}_L$, $\mathbf{Y}_R$, $\mathbf{y}_L$ and $\mathbf{y}_R$
3:     for s ← 1 to M do // for each source
4:         $\mathbf{y}_L[o]$ += $\mathbf{Y}_L[o]$
5:         $\mathbf{y}_R[o]$ += $\mathbf{Y}_R[o]$
6:         $o$ += 4L
7:     end for
8:     return $\mathbf{y}$
9: end function
```

output vectors $\mathbf{fy}$ (left $\mathbf{fy}_L$ and right $\mathbf{fy}_R$). To this end, we launch $4L$ *work-items*. Each *work-item* multiplies and reduces four real elements to two, one per channel. Algorithm 5 shows the main steps of this kernel.

**Algorithm 5** Fading of both source positions

```
1: function KFADE($\mathbf{y}$, $\mathbf{f}$, $\mathbf{g}$, $L$)
2:     $o$ = offset in $\mathbf{y}_L$, $\mathbf{y}_R$, $\mathbf{f}$ and $\mathbf{g}$
3:     $\mathbf{fy}_L[oy] = \mathbf{y}_L[oy]$ * $\mathbf{f}[of]$ + $\mathbf{y}_L[oy+2L]$ * $\mathbf{g}[of]$
4:     $\mathbf{fy}_R[oy] = \mathbf{y}_R[oy]$ * $\mathbf{f}[of]$ + $\mathbf{y}_R[oy+2L]$ * $\mathbf{g}[of]$
5:     return $\mathbf{fy}$
6: end function
```

The vectors $\mathbf{x}_{\mathcal{B}_i}$ and the weight factors $\{w_A, w_B, w_C, w_D\}$ of each sound source are transferred from CPU to GPU each time we receive a buffer of samples. However, Matrix $\mathbf{H}$, which contains the filters of the HRTF, together with the vectors $\mathbf{f}$ and $\mathbf{g}$, are transferred to the GPU before the application starts to process the audio samples (off-line).

### 4.3.2 Second Approach

We can add another level of parallelism to Algorithm 1 by distributing the computation across the two devices of the GPU. We can launch in parallel the same or different kernels to distinct OpenCL devices by associating a command-queue to each device, either sharing a context or having one context per queue. In our case, we have created one context per device, because the clFFT library only works with that configuration, and because if we spawn several threads, the OpenCL host binding is not guaranteed to be thread-safe. We can combine OpenCL with OpenMP, and execute one thread per device in the host. Each OpenMP thread can then be run in parallel and dispatch the kernels to a different device using the corresponding command-queue.

We have implemented two versions of the algorithm that use two devices in parallel to perform the audio processing: `2devmono` and `2devdual`. In both versions we start by scattering the input-vectors $\mathbf{x}_{\mathcal{B}_i}$, corresponding to different sources, to the two devices. Then each thread employs the proper routine from the clFFT library to perform in parallel the FFT of its sources in one of the devices.

Next, for Algorithm `2devmono`, we transfer the results of one device to the other using the host. From that point on, each device can process in parallel one of the channels: left and right. To that end, we have implemented a version of kernels `kBuildFilter`, `kMult`, `kRed` and `kFade`, where each *work-item* performs the computations with only one channel.

One of the problems of Algorithm `2devmono` is the communication cost. If we distribute half of the sources to each device, we have to transfer $ML$ complex elements from each device to the host and then from the host to the other device. An additional problem of this version of the algorithm is that once the FFT is performed, we associate the same computational load to each device. Therefore, we cannot balance the load when dealing with heterogeneous devices such as those comprised by the Mali GPU.

Algorithm `2devdual` tackles both problems. After performing the initial FFT in parallel, each device can use kernels `kBuildFilters`, `kMult` and `kRed` to obtain a partial version of buffer $\mathbf{y}$. This buffer reflects the effect of filtering only the sources stored on each device. In order to combine both partial results in parallel, each device transfers to the other the elements corresponding to one of the channels $\mathbf{y}_L$ or $\mathbf{y}_R$. Then both devices run in parallel an additional kernel `kAdd` to add the elements of both partial buffers. From this point on, both devices perform steps 7–9 of Algorithm 1 in parallel, but working only with one of the channels. That is, device 0 computes $\mathbf{fy}_L$ while device 1 computes $\mathbf{fy}_R$. Note that this version of the algorithm reduces the communication cost as we only have to transfer $2L$ elements between the devices. Moreover, except for the last steps of the algorithm, we can balance the computational workload in heterogeneous devices by appropriately distributing a different number of sources to each of them.

Using the same configuration with two contexts and one queue associated to each device we can exploit the asynchronous call mechanism provided by OpenCL. That is, we can launch each kernel to a different device without blocking the host thread that enqueues it. We have implemented a version of the algorithm, `2devasyn`, that uses the same parallelization strategy that algorithm `2devdual`, but exploiting this calling mechanism to dispatch each kernel to both devices.

## 5 Experimental evaluation

Our experiments evaluate the computational performance, energy consumption and energy efficiency of the HSA codes on the Exynos 5422 SoC, taking as a reference the time offered by an audio card from a mobile device. This specific audio application provides one frame with $L = 1,024$ samples per channel every $t_{\mathcal{B}} = 23.22$ ms (for a sample frequency $f_s = 44.1$ KHz). Assuming that our spatial audio application has to synthesize $M$ sound sources, we define $t_{\mathcal{P}}$ 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 will operate in real time provided $t_{\mathcal{P}} < t_{\mathcal{B}}$.

Since we target the HSA application, we adopt as a performance metric the maximum number of sound sources that can be rendered in real time in order to exhaust the processing on the Exynos 5422.

We are not only interested in maximizing the number of sound sources that can be rendered in real time, but also in reducing the energy consumption of the system. Therefore, we will analyze the energy consumed by the different configurations in order to maximize the sources-per-Watt that can be rendered in real time. In order to reduce the energy required by the algorithms, we will investigate the effect of changing the operating frequency of both the CPU cores and the GPU.

## 5.1 Performance and energy using the ARMv7 cores

In this section we evaluate the implementation that exploits the ARM cores of the Exynos 5442. Figure 5 shows the execution time obtained using both types of ARM cores, Cortex-A7 and Cortex-A15, for one and four cores. We can observe that the processing cost $t_{\mathcal{P}}$ increases linearly with the number of sources, with the addition of more cores allowing to manage a significantly higher number of sound sources. These experiments were carried out by setting both the Cortex-A7 and Cortex-A15 to their maximum clock frequencies: 1.4 GHz and 2.0 GHz respectively. The line corresponding to one Cortex-A15 core shows that it is possible to handle up to 177 sources in real time, while using four cores, this quantity increases to 361 sound sources. Figure 5 also illustrates that the Cortex-A15 cores are much faster than their Cortex-A7 counterparts. For example, using one Cortex-A7 core, we can only process 47 sound sources in real time, that is, 3.7 times less sources that with one Cortex-A15 core.

Regarding the energy efficiency of the algorithms, the first row of plots in Figure 6 shows the energy consumed to process the maximum number of sound sources that can be handled for each configuration in real time. We can observe there the variation of the consumption when we increase the frequency of the processors and the number of cores for both types of cores. An aspect worth mentioning in these results is that the Cortex-A7 cores consume significantly less energy than the Cortex-A15 cores, and that the gap between both types of cores increases with the frequency. It is also interesting to point out that, for both types of cores, the energy consumption increases very slowly with the number of cores.

Nevertheless, as we can appreciate in Figure 5, the maximum number of sources that can be processed in real time greatly varies for different configurations. Therefore, in order to offer a fair comparison of the energy consumption, the second row of plots in Figure 6 shows the maximum number of sources-per-Watt for each case. Using this metric, we can observe that the Cortex-A15 cores are only slightly more efficient than the Cortex-A7 cores. However, in the first case we obtain the highest efficiency with three cores at a frequency between

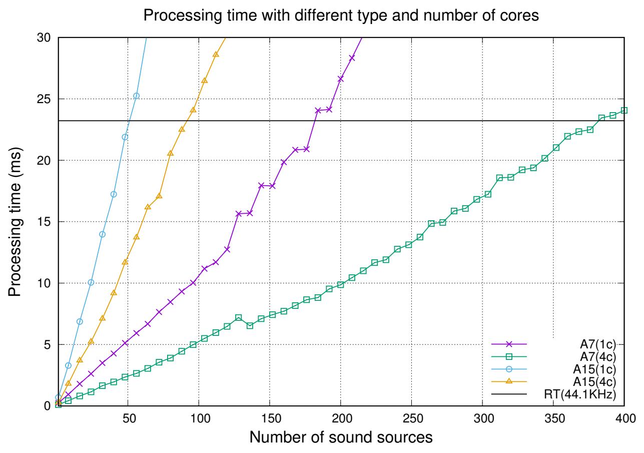


Fig. 5: Processing time using one and four ARM Cortex-A15 and Cortex-A7 cores. The black horizontal line marks the threshold that indicates that the application runs in real time ($t_{\mathcal{B}} = 23.22$ ms).

600 MHz and 1.0 GHz, while in the second case the best results are obtained using four cores at their highest frequencies.

## 5.2 Performance and energy using the Mali-T628 GPU

We next compare the performance of Algorithm 1 in the configurations shown in Table 1, which correspond to the different versions described in Section 4.

Tab. 1: Configurations for running Algorithm 1 in the Mali-T628 GPU.

| Name of the configuration | Number of devices | Number of GPU cores | CPU cores |
|---|---|---|---|
| `1dev0` | 1 | 4 | 1 |
| `1dev1` | 1 | 2 | 1 |
| `2devmono` | 2 | 4 + 2 | 2 |
| `2devdual` | 2 | 4 + 2 | 2 |
| `2devasyn` | 2 | 4 + 2 | 1 |

Figure 7 shows the execution time of the different configurations as we increase the number of sound sources to render. As expected, the performance is lower when we use the GPU device with two computational units (`1dev1`). Furthermore, as the number of sound sources increases, the gap between the processing times $t_{\mathcal{P}}$ of both devices increases significantly (`1dev0` vs. `1dev1`).

The best results are obtained with the `2devdual` version of the algorithm, which is only improved by the `1dev0` when handling a few sources ($< 15$). Contrarily, even using the two devices, the `2devmono` version is slightly worse

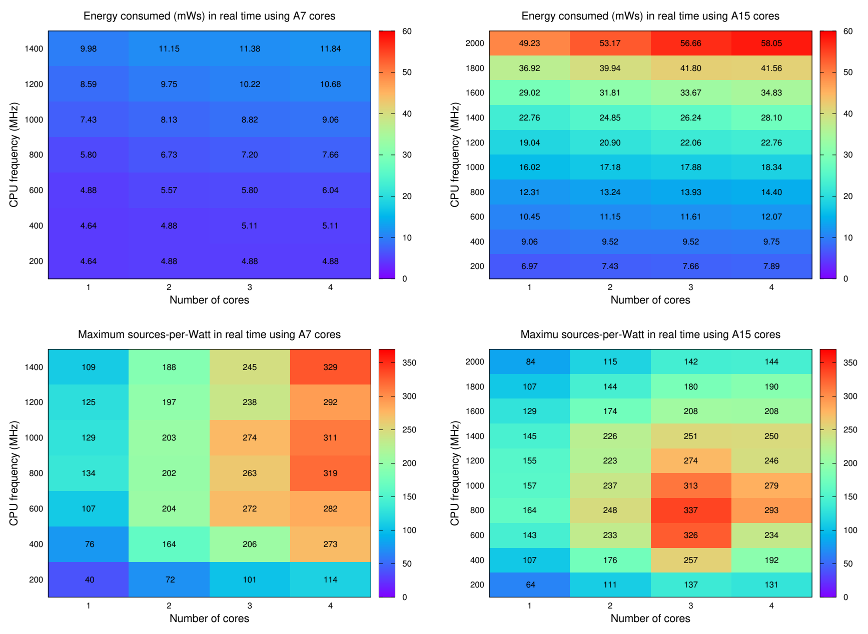


Fig. 6: Energy consumption and efficiency of the HSA codes on the Exynos 5422 system-on-chip (SoC). The plots in the left- and right-hand side columns correspond to the results on the ARM Cortex-A7 and Cortex-A15 respectively. The first row of plots contains the energy consumed to process the maximum number of sound sources in real time. The second row illustrates the sources-per-Watt (energy efficiency) that the algorithm can handle in real time.

than the version using only the device with four compute units. This is due to the communication cost and load unbalance of the algorithm already commented in Section 4. The good results offered by the `2devdual` version demonstrate that it reduces the effect of both problems. We have applied a balancing technique that maps the computations corresponding to 2/3 of the sources to the device with four compute units and the remaining 1/3 to the device with two compute units in order to provide a similar computational load to each of the 6 compute units. Finally, the results obtained with the `2devasync` version demonstrate that using one CPU host thread and the asynchronous call mechanism provided by OpenCL is less effective to deal with the two devices in parallel that using two OpenMP threads, especially when we have to process less than 70 sources. Two OpenMP threads can leverage two of the CPU cores in parallel to transfer the data and launch the kernels to each GPU device, whereas one thread uses only one CPU core to prepare and launch sequentially the kernels to each device. Even if the launch call returns immediately after the command is enqueued, and likely before the kernel has even started execution, this version is less efficient

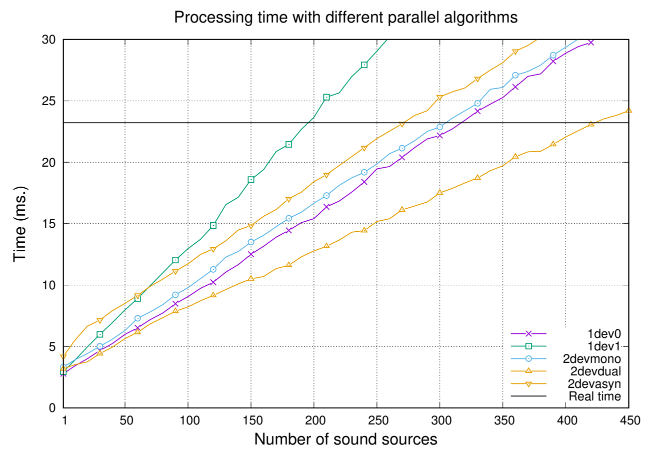


Fig. 7: Execution time of the parallel algorithms increasing the number of sources. The horizontal line marks the threshold between a real-time application and an off-line technology.

than the one combining OpenCL and OpenMP. Therefore, in the rest of the experimental analysis, we will always use the results obtained with the `2devdual` version of the algorithm.

In the GPU-based implementation of Algorithm 1, most of the computations are performed in the Mali-T628. However, it is important to evaluate the influence of the type of CPU core that is used to transfer data to the GPU and dispatch the different kernels. Figure 8 shows the highest number of sound sources that the configuration `2devdual` can handle in real time for each of the seven work frequencies of the Mali-T628 GPU, using either the Cortex-A7 and Cortex-A15 cores as host CPU and with three different frequencies on each type of core. Our experiments show that the number of sources that are processed clearly increases when we use the faster Cortex-A15 cores as the host CPU. This is mainly due to differences in the communication time, which depends on the type of CPU core that is in charge of transferring the data. Our profiling of the execution shows that the running time of the kernels on the device does not depend on the CPU core. However, the time the kernels spend enqueued as well as the time they wait between the moment they are submitted to the device to the moment the execution actually commences, clearly differs depending on the type of cores: Cortex-A7 or Cortex-A15.

The maximum number of sources slowly increases with the frequency of the GPU for both kinds of host cores and all the CPU frequencies. Besides, there is a large gap in the number of sources that can be processed when we increase the CPU frequency from 400 to 800 MHz. However, this difference is narrower when we raise the frequency from 800 MHz to 1.4 GHz.

In order to analyze the energy consumption of the `2devdual` configuration, we measured the power dissipation of the platform for different combinations

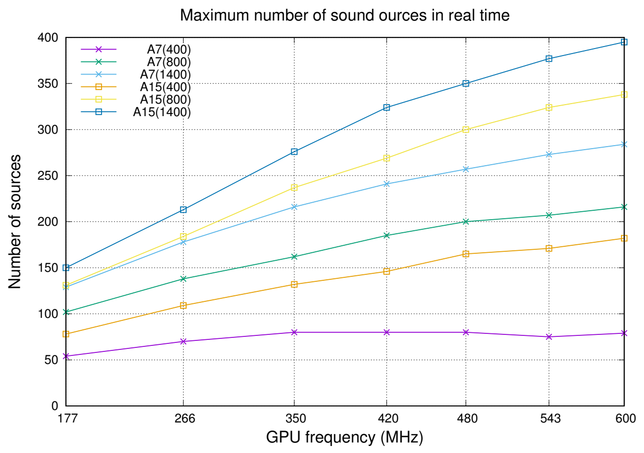


Fig. 8: Maximum number of sound sources that can be processed in real time varying the frequency of the Mali-T628 GPU. The different lines correspond to the results using as host CPU both types of ARM cores (Cortex-A7 and Cortex-A15) at different frequencies.

of CPU and GPU frequencies. The first row of plots in Figure 9 shows the results obtained using the two devices of the GPU and either a Cortex-A7 core or a Cortex-A15 core. The energy consumption increases at a very similar pace using both kinds of cores, with the Cortex-A15 presenting the highest energy consumption.

The second row of plots in Figure 9 shows the sources-per-Watt that can be processed in the same cases. We observe that the program cannot process even a single sound source in real time when we use the lowest CPU frequency for both kinds of CPU regardless of the GPU frequency. The best results are obtained in an area defined by the combination of GPU frequencies in the range 350 to 480 MHz with CPU frequencies in between 600 MHz and 1.4 GHz, both for the ARM Cortex-A7 and Cortex-A15. Additionally, for most combinations of GPU frequencies with CPU frequencies, the configurations that involve the Cortex-A7 overcome their counterparts with the Cortex-A15 in terms of energy consumption. As shown in Figure 8, using the Cortex-A7 cores, we can process less sound sources in real time, but as they dissipate less power, they can handle more sources-per-Watt.

## 5.3 Best configuration

There are different criteria to determine which is the configuration that obtains the best performance in the Exynos 5422 SoC. This depends on the target of the application. For example, the goal may be to maximize the number of sources in real time, without taking into account the energy consumed. Alternatively one can aim to minimize the energy consumption to process the audio frames

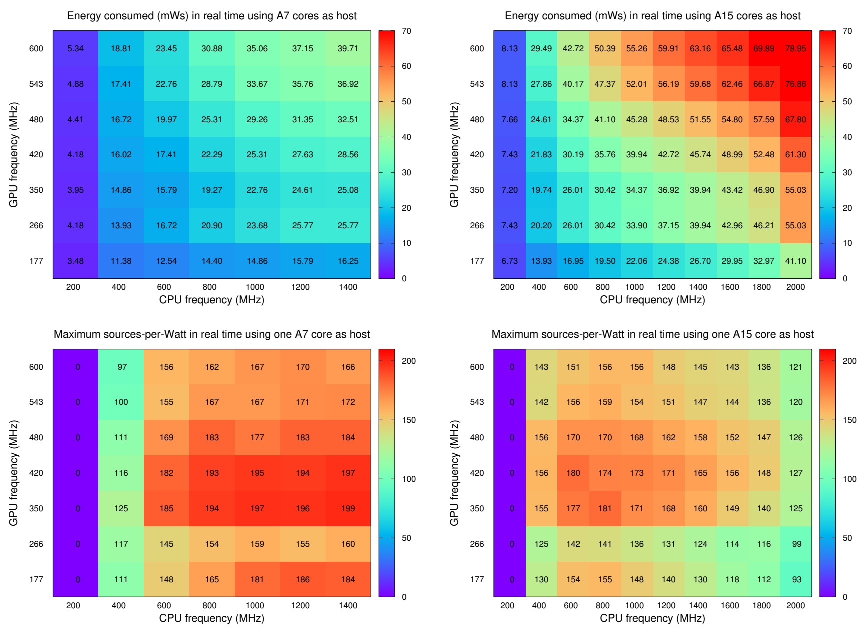


Fig. 9: Energy consumption and efficiency of the configuration `1dev0` combining different CPU frequencies with the GPU frequencies. The plots in the left- and right-hand side columns correspond to the results using Cortex-A7 and Cortex-A15 as host CPU core, respectively. The first row of plots contains the energy consumed to process the maximum number of sound sources in real time. The second row illustrates the sources-per-Watt (energy efficiency) that can be handled in real time.

corresponding to a fixed number of sound sources in a real-time scenario.

Figure 10 analyzes several configurations that maximize the number of sources that can be rendered in real time. To this end, in the case of the configurations implemented on the CPU cores, we use all Cortex-A7 cores and three or four Cortex-A15 cores depending on the CPU frequency. The combination of the GPU, running at the highest frequency, together with two Cortex-A15 achieves the highest performance except with the lowest CPU frequency. The maximum number of sound sources is achieved with this configuration with the highest CPU frequency: 412 sound sources rendered in real time.

Focusing on energy efficiency, we leverage the sources-per-Watt metric to analyze the effect of CPU frequency scaling. Figure 11 shows that the highest efficiency is obtained by a configuration with three Cortex-A15 cores at a frequency of 800 MHz. With this configuration, we can handle 201 sound sources in real time consuming 13.85 mWs.

We also analyze the performance of the proposed configurations by fixing the amount of sound sources, and then optimize the energy efficiency. Figure 12

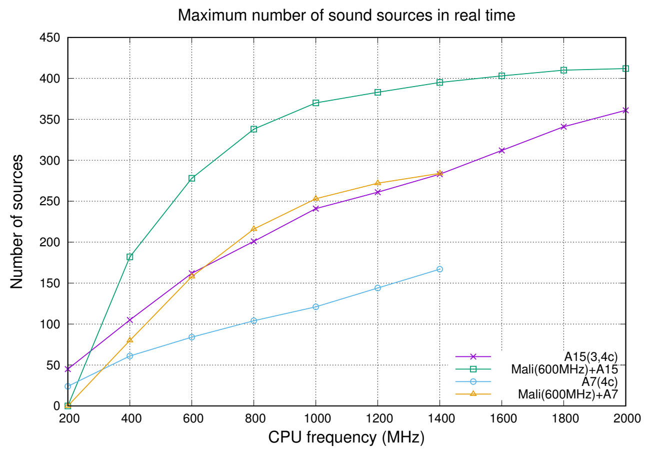


Fig. 10: Maximum number of sound sources than can be rendered in real time varying the CPU frequencies for the Cortex-A7 and Cortex-A15, with the Mali-T628 GPU running at its highest frequency (600 MHz).

shows the processing time of different configurations to render 20 sound sources. The results are shown varying the CPU frequencies of both Cortex-A15 and Cortex-A7 and using the number of cores and GPU frequency that achieves the lowest execution times. All algorithms compute 20 sound sources in real time at all CPU frequencies, except the configurations composed of the GPU and either the Cortex-A7 or the Cortex-A15 cores at their lowest frequency (200 MHz). The fastest configuration at all CPU frequencies is the one using three Cortex-A15 cores. The best case is obtained with the CPU frequency set to 2 GHz and three Cortex-A15 cores, which is able to render 20 sound sources in 0.90 milliseconds.

If our goal is to invest the minimum energy to render 20 sound sources, Figure 13 shows that the best results by far are always obtained with the configurations that employ both types of CPU cores. Using CPU frequencies between 400 MHz and 1.8 GHz, when available, both configurations consume less than 2.0 mWs to process a single frame for each of the 20 sound sources in real time. The time spent to process the frames depends on the configuration and is reported in Figure 12. Therefore, if we want to combine fast sound rendering with low energy consumption we should use the CPU algorithm with three Cortex-A15 cores.

To put these results into perspective, the Samsung Galaxy S5 G900H cell phone includes our target processor Exynos 5422. This mobile device uses a Lithium-Ion battery with a capacity of 2.8 Ah and a voltage of 3.85V. Therefore, this battery provides a maximum of 10.78 Wh. Let us assume that we are devoting the battery only to sustain the execution of our application rendering 20 sound sources in real time. That is, let us assume that we are processing one frame for each of the 20 sources every 23.22 ms, despite for almost all con-

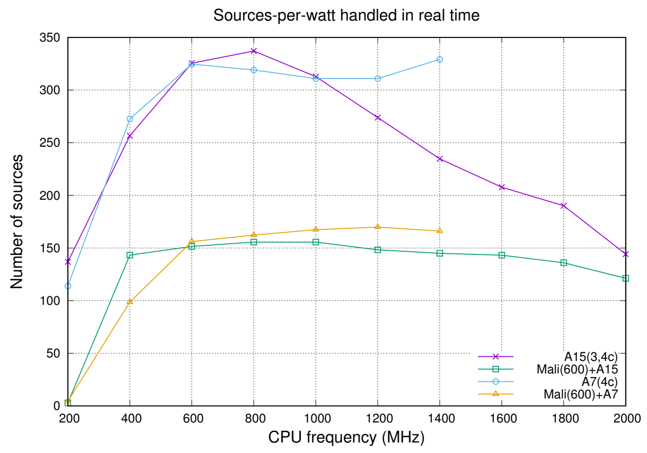


Fig. 11: Number of sound sources-per-Watt that can be processed in real time varying the CPU frequencies. The different lines correspond to the results obtained using different combinations of the CPU cores and the Mali-T628 GPU.

figurations, a much shorter time is required to complete the processing. In the most efficient configuration, using the four cores of the Cortex-A7 at 1.0 GHz, we can run our application for around 191 hours, as in this case we are consuming 1.31 mWs every 23.22 ms. In the best GPU configuration, that is, using the Mali-T628 at 177 MHz and two Cortex-A7 at 1.0 GHz, the application can be run for around 56 hours. In contrast, in the worst case of the four selected configurations, the GPU together with two Cortex-A15 at 2 GHz can barely run the application for around 24 hours before exhausting the battery. Figure 13 exposes the importance of exploring all computational resources integrated in the Exynos 5422 SoC. If we take as reference an straightforward OpenMP implementation that runs using all four A15 cores in parallel at their highest frequency, the battery will be exhausted after 80 hours. Therefore, our most efficient configuration can extend the autonomy of the device executing the HSA application with respect to that reference configuration by a 238%.

## 6 conclusions

In this paper, we have explored the capabilities of the Exynos 5422 architecture to tackle a spatial audio application (involving multiple convolutions). We have proposed efficient implementations that extract concurrency to feed the CPU processors via NEON instructions and OpenMP. Moreover, we have optimized this spatial audio application in the GPU Mali-T628 that is embedded in the Exynos 5422 processor, using OpenCL to leverage the heterogeneous devices.

We have evaluated the computational performance in terms of maximum number of sound sources that can be rendered in real time, as well as energy ef-

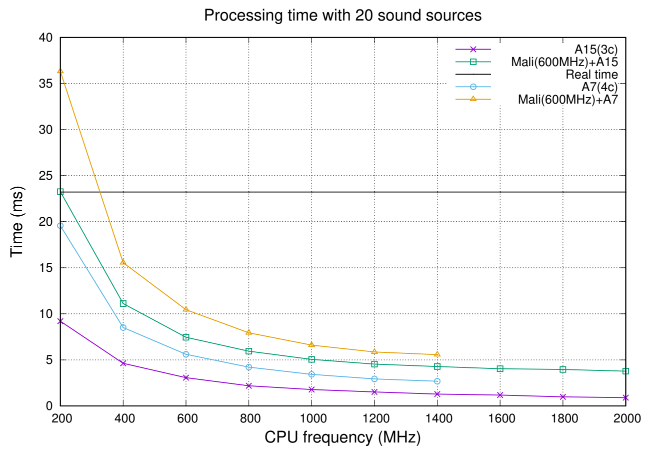


Fig. 12: Processing time in milliseconds for rendering 20 sound sources. The different lines correspond to the results obtained using different combinations of the CPU frequencies and a GPU frequency of 600 MHz. The horizontal line marks the threshold between a real time application and an off-line application.

ficiency in terms of maximum sources-per-Watt. Our experiments indicate that the optimal configuration depends on the purpose of the application. Specifically, if the purpose is to maximize the number of sound sources rendered in real time, independently of the energy consumption, we should rely on the OpenCL implementation that operates both GPU devices at their highest frequency (600 MHz) and two Cortex-A15 cores also operating at their highest frequency (2 GHz). In contrast, for energy efficiency, the maximum sound sources-per-Watt is obtained using three Cortex-A15 cores at 800 MHz. In case the application has to be executed on the GPU (because CPU is busy with other tasks), we should set the Mali GPU frequency to 350 MHz and employ two Cortex-A7 at their highest frequency.

Finally, we must highlight the dependency that exists between the battery lifetime and the use of different computational resources. For example, in case of reproducing 20 sound sources, the best option, both in terms of real-time processing and energy consumption, is to use the configuration that exploits the 3 Cortex-A15 cores at 1.8 or 2 GHz.

This work points out the necessity of properly analyzing the different alternatives for implementing an application in this type of low-power SoC processors. Exploiting in an optimal way the capabilities of the platform we can extend the autonomy to execute a HSA application by a 238%.

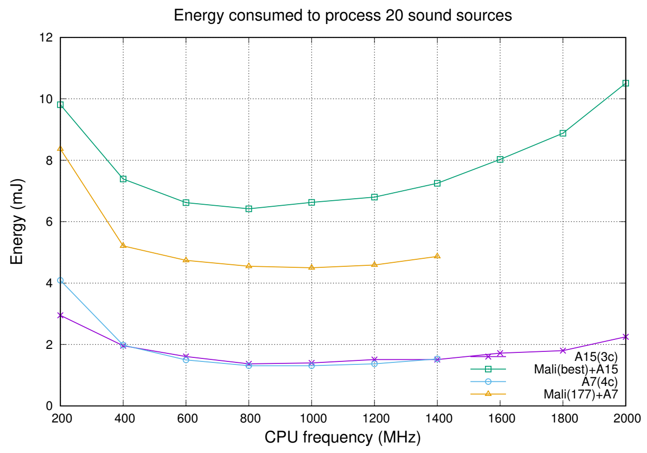


Fig. 13: Energy consumed for rendering one frame of 20 sound sources varying the CPU frequency. The different lines correspond to the results obtained using the best configurations of the algorithms using the Cortex cores and Mali-T628 GPU.

## Acknowledgments

We would like to thank the reviewers of this paper for their very insightful comments that helped us to greatly improve it. This work has been partially funded by Spanish Ministerio de Economía y Competitividad projects: TIN2014-53495-R and TEC2015-67387-C4-1-R, the University project UJI-B2016-20, together with the project PROMETEOII/2014/003. Dr. Jose A. Belloch is supported by GVA postdoctoral contract APOSTD/2016/069.

## References

[1] V. K. Parikh, P. T. Balsara, and O. E. Eliezer, "All digital-quadrature-modulator based wideband wireless transmitters," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 11, pp. 2487–2497, Nov 2009.

[2] A. K. Mustafa, S. Ahmed, and M. Faulkner, "Bandwidth limitation for the constant envelope components of an OFDM signal in a LINC architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 9, pp. 2502–2510, Sept 2013.

[3] Z. Liu, K. Dickson, and J. V. McCanny, "Application-specific instruction set processor for SoC implementation of modern signal processing algorithms," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 52, no. 4, pp. 755–765, April 2005.

[4] J. Ramanujam, J. Hong, M. Kandemir, A. Narayan, and A. Agarwal, "Estimating and reducing the memory requirements of signal processing codes for embedded systems," *IEEE Transactions on Signal Processing*, vol. 54, no. 1, pp. 286–294, Jan 2006.

[5] O. Chen, M. Hsia, and C. Chen, "Low-complexity inverse transforms of video codecs in an embedded programmable platform," *IEEE Transactions on Multimedia*, vol. 13, no. 5, pp. 905–921, Oct 2011.

[6] J. Chen and K. Liu, "Low-power architectures for compressed domain video coding co-processor," *IEEE Transactions on Multimedia*, vol. 2, no. 2, pp. 111–128, Jun 2000.

[7] Y. Andreopoulos, D. Jiang, and A. Demosthenous, "Prediction-based incremental refinement for binomially-factorized discrete wavelet transforms," *IEEE Transactions on Signal Processing*, vol. 58, no. 8, pp. 4441–4447, Aug 2010.

[8] S. Ren, N. Deligiannis, Y. Andreopoulos, M. A. Islam, and M. van der Schaar, "Dynamic scheduling for energy minimization in delay-sensitive stream mining," *IEEE Transactions on Signal Processing*, vol. 62, no. 20, pp. 5439–5448, Oct 2014.

[9] Y. Keller, R. R. Coifman, S. Lafon, and S. W. Zucker, "Audio-visual group recognition using diffusion maps," *IEEE Transactions on Signal Processing*, vol. 58, no. 1, pp. 403–413, Jan 2010.

[10] S. D. Larbi and M. Jaidane-Saidane, "Audio watermarking: a way to stationnarize audio signals," *IEEE Transactions on Signal Processing*, vol. 53, no. 2, pp. 816–823, Feb 2005.

[11] "ARM NEON," https://developer.arm.com/technologies/neon, (accessed 2017 October 3).

[12] J. A. Belloch, M. Ferrer, A. Gonzalez, F. 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, Jul 2013.

[13] D. N. Zotkin, R. Duraiswami, and L. S. Davis, "Rendering localized spatial audio in a virtual auditory space," *IEEE Transactions on Multimedia*, vol. 6, no. 4, pp. 553–564, Aug 2004.

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

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

[16] Z. Foo, D. Devescery, M. H. Ghaed, I. Lee, A. Madhavan, Y. S. Park, A. S. Rao, Z. Renner, N. E. Roberts, A. D. Schulman, V. S. Vinay, M. Wieckowski, D. Yoon, C. Schmidt, T. Schmid, P. Dutta, P. M. Chen, and D. Blaauw, "A low-cost audio computer for information dissemination among illiterate people groups," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 8, pp. 2039–2050, Aug 2013.

[17] G. Mitra, B. Johnston, A. Rendell, E. McCreath, and J. Zhou, "Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms," in *Proc. IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, Boston, USA, May 2013, pp. 1107–1116.

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

[19] R. Wang, J. Wan, W. Wang, Z. Wang, S. Dong, and W. Gao, "High definition IEEE AVS decoder on ARM NEON platform," in *Proc. 20th IEEE International Conference on Image Processing (ICIP)*, Melbourne, Australia, Sept 2013, pp. 1524–1527.

[20] Z. Ma, H. Hu, and Y. Wang, "On complexity modeling of H.264/AVC video decoding and its application for energy efficient decoding," *IEEE Transactions on Multimedia*, vol. 13, no. 6, pp. 1240–1255, Dec 2011.

[21] N. Mastronarde, K. Kanoun, D. Atienza, P. Frossard, and M. van der Schaar, "Markov decision process based energy-efficient on-line scheduling for slice-parallel video decoders on multicore systems," *IEEE Transactions on Multimedia*, vol. 15, no. 2, pp. 268–278, Feb 2013.

[22] S. Holgersson, "Optimising IIR filters using ARM NEON," Master's thesis, University of Denmark, 2012.

[23] J. A. Belloch, F. J. Alventosa, P. Alonso, E. S. Quintana-Ortí, and A. M. Vidal, "Accelerating multi-channel filtering of audio signal on ARM processors," *Journal of Supercomputing*, vol. 73, no. 1, pp. 203–214, Mar 2017.

[24] J. A. Belloch, A. Gonzalez, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, "Vectorization of binaural sound virtualization on the ARM cortex-A15 architecture," in *Proc. 23rd European Signal Processing conference, (EUSIPCO)*, Nize, France, Sep 2015, pp. 1601–1605.

[25] D. R. Begault, *3-D sound for virtual reality and multimedia.* San Diego, CA, USA: Academic Press Professional, Inc., 1994.

[26] "Listen HRTF database," *online at: http://recherche.ircam.fr/equipes/salles/listen/index.html.* (accessed 2017 Oct 3)

[27] A. Kudo, H. Hokari, and S. 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.

[28] S. Barrachina, M. Barreda, S. Catalán, M. S. Dolz, G. Fabregat, R. Mayo, and E. S. Quintana-Ortí, "An integrated framework for power-performance analysis of parallel scientific workloads," in *Proc. 3rd International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY) (2013)*, Lisbon, Portugal, Mar 2013, pp. 114–119.

[29] R. Gensh, A. Aalsaud, A. Rafief, F. Xia, A. Iliasov, A. Romanovsky, and A. Yakovlev, "Experiments with the Odroid-XU3 board," Newcastle University, Computing Science, Newcastle upon Tyne, Tech. Rep. CS-TR-1471, May 2015.

[30] "ARMv7 NEON data types," https://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/ARM-NEON-Intrinsics.html, (accessed 2017 Oct 3).

[31] M. Scarpino, *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning, 2012.

[32] E. C. Lfeachor and B. W. Jervis, *Digital signal processing: a practical approach*, Prentice Hall, 2002.

[33] A. V. Oppenheim, A. S. Willsky, and S. Hamid, "Signals and systems," ser. Processing series. Prentice Hall, 1997.

[34] "Fast Fourier Transform West," http://www.fftw.org, (accessed 2017 Oct 3).

[35] "OpenCL fast fourier transforms," http://clmathlibraries.github.io/clFFT, (accessed 2017 Oct 3).