Multicore implementation of a multichannel parallel graphic equalizer

Jose A. Belloch[1] (ID) **· José M Badía[2] · German León[2] · Balázs Bank[3] · Vesa Välimäki[4]**

Abstract

Numerous signal processing applications are emerging on mobile computing systems. These applications are subject to responsiveness constraints for user interactivity and, at the same time, must be optimized for energy efficiency. Many current embedded devices are composed of low-power multicore processors that offer a good trade-off between computational capacity and low power consumption. In this context, equalizers are widely used in multiple mobile-based applications such as "Music streaming" to adjust the levels of bass and treble in sound reproduction. In this study, we evaluate a graphic equalizer from audio, computational capacity, and energy efficiency perspectives, as well as the execution of multiple real-time equalizers running on an embedded quad-core processor of a mobile device. To this end, we experiment with the working frequencies as well as the parallelism that can be extracted from a quad-core ARM Cortex-A57. Results show that using high CPU frequencies and three or four cores, our parallel algorithm is able to equalize more than five channels per watt in real time with an audio buffer of 4096 samples, which implies a latency of 92.8 ms at the standard sample rate of 44.1 kHz.

Keywords Audio systems · Real time · Embedded systems · System-on-chip (SoC)

1 Introduction

Low-power (embedded) processors play an important role for a myriad of signal processing applications, such as communications [1], image processing [2], visual detection [3], speech recognition [4], and audio processing [5], among others. In the era of smartphones and tablets, these energy-efficient architectures have increased significantly their computational capacity and are nowadays utilized in a large volume of multimedia, including video and audio processing.

✉ Jose A. Belloch
jbelloc@ing.uc3m.es

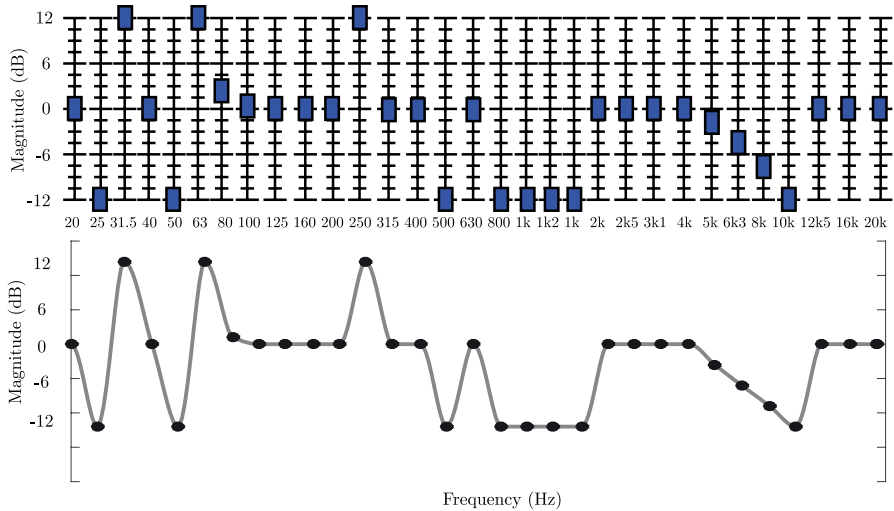Extended author information available on the last page of the article

Fig. 1 At the top: an example of sliders positions. At the bottom: Magnitude response approximated by the graphic equalizer with these sliders positions

In this context, equalizers are widely used in multiple mobile-based applications such as "Music streaming" to adjust the levels of bass and treble in sound reproduction [6]. In fact, equalizing filters are used for improving the frequency response of loudspeakers or headphones [7–9] and for reducing the effects of room acoustics on the sound quality [10, 11]. A graphic equalizer consists of many filters with fixed center frequencies, and the gain of each filter, which is often called the *command gain*, is the only adjustable parameter [12], as shown in Fig. 1. A graphic equalizer can be implemented as a cascade of equalizing filters [12, 13] or as a parallel bank of bandpass filters [12, 14]. The parallel structure is more advantageous compared to the series one in terms of quantization noise performance [15], and also supports the parallel computation of the filter sections, leading to a performance benefit on GPUs, for example [16–18]. In this study, we evaluate the parallel graphic equalizer (PGE). Note however that here we are not utilizing the parallel structure of the filter for parallel computation. On the contrary, the parallellization comes from computing multiple channels in the same time. This also means that our findings are applicable to series equalizers as well, or any other real-time audio algorithms that should be computed for multiple channels.

However, the parallel graphic equalizer (PGE) has the disadvantage that updating its parameters requires about hundred times more operations than the update in a standard equalizer [14]. A parameter update is necessary during interactive operation every time a command gain is modified. When the parameters are redesigned, a complex target response must be produced from the command gain values. This has been made more efficient by computing the target response based on minimum-phase basis functions and a more efficient WLS design in [19].

Nowadays, there exists system-on-chip (SoC) that delivers notable computational capacity while partially retaining the appealing low power consumption. One

example of this type of system is the NVIDIA Jetson Nano board [20] which is based on a quad-core ARM Cortex-A57 CPU at 1.43 GHz with 4 GiB of LPDDR4 memory, and a 128-core Nvidia Maxwell GPU. One of its main features is its low power consumption combined with several levels of parallelism yielding a very high performance per Watt. Moreover, it also offers the possibility of adjusting its energy consumption by reducing the frequency of the CPU cores or the GPU.

Up to now, there have been studies analyzing the computational performance of filtering process of the PGE without taking into account the mandatory updating process that occurs when the sliders changes (see Fig. 1) [18], as well as works implementing this updating process in a sequential way [21]. In this work, we analyze the performance of the whole system in terms of audio processing and energy consumption as a function of CPU operating frequency and number of cores used. The aim of this work is to find an efficient implementation which exhibits a proper trade-off between performance and energy consumption taking into account the real-time constraint.

This paper is structured as follows. Section 2 offers a brief overview of the Parallel Graphic Equalizer. Section 3 describes the multicore implementation. Section 4 explores the performance of the PGE in terms of maximum number of audio channels that can be rendered in real time; provides a detailed analysis of the power dissipation; and analyzes the energy efficiency of different hardware configurations. Finally, Sect. 5 closes the paper with concluding remarks.

2 Parallel graphic equalizer

Equalizers correct or enhance the spectrum of a signal in order to meet a desired requirement. Equalizers are widely used in music production and in sound reproduction to control the timbral balance of music [22, 23], as well as to reduce the effects of room acoustics on the sound quality [10]. In graphic equalizers, the user controls the gain of each frequency band using a set of sliders that modify the desired magnitude response [12, 14, 24–26]. Common graphic equalizers control the gain at 31 frequency bands spaced one third of an octave apart.

The basic idea of the PGE [14, 19] is that based on the slider positions set by the user (top plot in Fig. 1), a smooth target frequency response $H_t(\vartheta_n)$ is computed, where ϑ_n ($n = 1, 2, ..., N$) represents a finite set of angular frequencies. Then, the next step is the parameter estimation for the parallel IIR filters. The filter structure is composed of a set of parallel second-order sections having the transfer function

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

where d_0 is called the direct path gain and K is the number of filter sections.

The first task in fixed-pole parallel filter design is setting the pole positions, which control the frequency resolution of the design [27]. It was shown in [14] that having two times as many pole pairs as command points and placing them logarithmically in frequency provides a sufficient resolution. The pole radii $|p_k|$ are set such

that the magnitude responses of the parallel sections meet approximately at their -3-dB points [14, 27]. The transfer function (1) becomes linear in its free numerator parameters $b_{k,0}$, $b_{k,1}$, and d_0, when the denominator coefficients are determined by the fixed poles. Writing (1) in matrix form yields

$$\mathbf{h} = \mathbf{Mp}, \tag{2}$$

where $\mathbf{h} = [H(\vartheta_1) \dots H(\vartheta_N)]^T$ is a column vector containing the resulting frequency response of the parallel filter, $\mathbf{p} = [b_{1,0}, b_{1,1}, \dots, b_{K,0}, b_{K,1}, d_0]^T$ is a column vector containing the free parameters, and $\mathbf{M}$ is a modeling matrix, which contains the sampled frequency responses of the second-order all-pole filters $1/(1 + a_{k,1}e^{-j\vartheta_n} + a_{k,2}e^{-j2\vartheta_n})$ in its odd columns, their delayed versions (multiplied by $e^{-j\vartheta_n}$) in the even columns, and last, a column of 1's, which corresponds to the constant frequency response of the direct path. As in [14], we use a frequency-dependent weighting for LS error minimization [28, 29], in the form of a diagonal matrix $\mathbf{W}$ whose values are computed by the the weighting function $W(\vartheta_n) = 1/|H_t(\vartheta_n)|$, where $W(\vartheta_n)$ is a real-valued non-negative weight at frequency ϑ_n.

Finally, we compute the optimal filter parameters $\mathbf{p}_{\text{opt}}$. Note that this computation must be carried out each time the sliders change, since $\mathbf{h}$ and $\mathbf{W}$ are updated based on the slider positions:

$$\mathbf{p}_{\text{opt}} = (\mathbf{M}^T\mathbf{W}^T\mathbf{WM})^{-1}\mathbf{M}^T\mathbf{W}^T\mathbf{Wh}. \tag{3}$$

To this end, we use the **QR** decomposition of the matrix $\mathbf{M} = \mathbf{QR}$, where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix. Thus, the pseudo-inverse is computed as

$$
\begin{aligned}
(\mathbf{WM})^+ &= (\mathbf{M}^T\mathbf{W}^T\mathbf{WM})^{-1}(\mathbf{W}^T\mathbf{M}^T) \\
&= (\mathbf{R}^T\mathbf{Q}^T\mathbf{W}^T\mathbf{WQR})^{-1}(\mathbf{R}^T\mathbf{Q}^T\mathbf{W}^T) \\
&= (\mathbf{R}_1^T\mathbf{E}^T\mathbf{ER}_1)^{-1}(\mathbf{R}_1^T\mathbf{E}^T) \\
&= \mathbf{R}_1^{-1}(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{R}_1^{-T}\mathbf{R}_1^T\mathbf{E}^T \\
&= \mathbf{R}_1^{-1}(\mathbf{E}^T\mathbf{E})^{-1}\mathbf{E}^T,
\end{aligned}
$$

where $\mathbf{E} = \mathbf{WQ}_1$. Note that $\mathbf{R} = [\mathbf{R}_1^T \quad \mathbf{0}]^T$ and $\mathbf{Q} = [\mathbf{Q}_1 \quad \mathbf{Q}_2]$. Thus, we can discard those elements of $\mathbf{Q}$ that are multiplied by zeros. In order to obtain $\mathbf{E}^T\mathbf{E}$, we need to obtain a second **QR** decomposition:

$$\mathbf{E}^T\mathbf{E} = \mathbf{R}_E^T\mathbf{Q}_E^T\mathbf{Q}_E\mathbf{R}_E = \mathbf{R}_{E1}^T\mathbf{R}_{E1}, \tag{4}$$

where $\mathbf{R}_E = [\mathbf{R}_{E1}^T \quad \mathbf{0}]^T$. Now we can compute $\mathbf{p}_{\text{opt}}$ as

$$
\begin{aligned}
\mathbf{p}_{\text{opt}} &= (\mathbf{WM})^+(\mathbf{Wh}) \\
\mathbf{p}_{\text{opt}} &= \mathbf{R}_1^{-1}(\mathbf{R}_{E1}^T\mathbf{R}_{E1})^{-1}\mathbf{E}^T(\mathbf{Wh}) \\
\mathbf{R}_{E1}^T\mathbf{R}_{E1}\mathbf{R}_1\mathbf{p}_{\text{opt}} &= \mathbf{E}^T(\mathbf{Wh}).
\end{aligned}
$$

Denoting $\mathbf{c} = \mathbf{E}^T \mathbf{W} \mathbf{h}$, $\mathbf{v} = \mathbf{R}_1 \mathbf{p}_{\mathrm{opt}}$, and $\mathbf{u} = \mathbf{R}_{E1} \mathbf{v}$, we have to solve three triangular linear systems to obtain $\mathbf{p}_{\mathrm{opt}}$:

$$\begin{aligned}
\mathbf{R}_{E1}^T \mathbf{u} &= \mathbf{c}, \\
\mathbf{R}_{E1} \mathbf{v} &= \mathbf{u}, \text{ and} \\
\mathbf{R}_1 \mathbf{p}_{\mathrm{opt}} &= \mathbf{v}.
\end{aligned} \tag{5}$$

3 Multicore implementation

The aim of this work is to analyze how an embedded quad-core processor is able to manage a multichannel parallel graphic equalizer. To this end, we leverage OpenMP [30] to parallelize both the computation of the filter coefficients by using the weighted least squares method, and the filtering process that will be done through each one of 62 second-order sections that compose each one of the channels to be processed, as shown in Fig. 2. Note that all channels can have different equalizer settings, so each channel needs to compute its coefficients separately. That means that different OpenMP threads can perform the whole steps of different equalization in parallel: for each equalization, the thread first computes
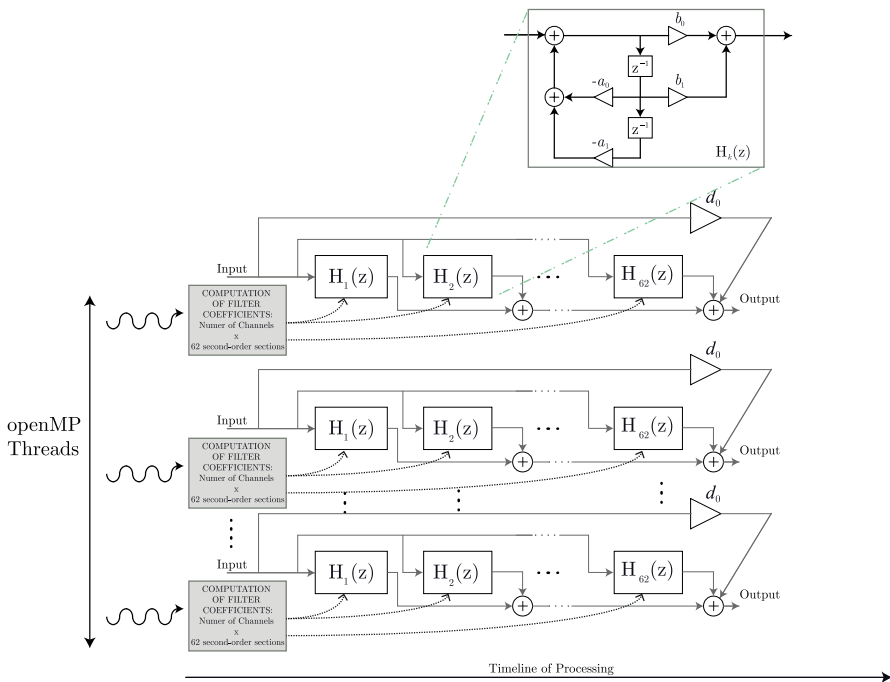

Fig. 2 Parallel implementation of the two steps of the multichannel parallel graphic equalizer: Computation of coefficients and filtering process

the coefficients of a filter and then sequentially executes the multiplications and sums that constitute the filtering operations of an IIR filter. Thus, the openMP threads that are shown at the bottom of Fig. 2 are the same that are shown at the top. In fact, the red arrow at the figure shows the flow of the execution. Note that if we had enough computational resources, we could perform all the equalizations in parallel without any kind of communication or synchronization among the OpenMP threads.

In order to execute the operations that update the coefficients of the filter, we use various software libraries for numerical linear algebra, such as LAPACK (*Linear Algebra Package*) [31] and BLAS (*Basic Linear Algebra Subprograms*) [32]. Since we are dealing with matrices and vectors of small sizes, instead of using the multi-threaded versions of these libraries, we use their sequential version in combination with the parallelism provided by OpenMP to equalize different channels in parallel. Table 1 shows the routines that have been used in order to carry out the updating computations of the filters coefficients.

3.1 Evaluation platform

We have performed our experiments in a Jetson Nano development kit, which is a platform launched by NVIDIA in 2019. It combines low cost, high computational performance and low power consumption. Our platform contains a quad-core ARM Cortex-A57 processor, an NVIDIA GPU with 128 cores and a Maxwell architecture, and four GiB of LPDRR4 memory, among other components to develop all kind of applications.

All our experiments were executed in the A57 multicore processor designed by ARM implementing the ARMv8-A 64-bit microarchitecture. It contains two MiB of L2 cache, 48 KiB of L1 instruction cache and 32 KiB of L1 data cache. Each core includes a vector floating point unit and supports the NEON SIMD extension.

Table 1 Routines of LAPACK and BLAS that have been used for implementing the coefficients update of the filters

Operations	Description	Routine
$\mathbf{E} \leftarrow \mathbf{W}\mathbf{Q}_1$	Weighting of matrix $\mathbf{Q}_1$. $\mathbf{W}$ is a diagonal matrix	–
$\mathbf{h} \leftarrow \mathbf{W}\mathbf{h}$	Element-wise multiplication $\mathbf{W}$ is a diagonal matrix	–
$\mathbf{c} \leftarrow \mathbf{E}^T\mathbf{h}$	Vector-Matrix Multiplication	`xgemv`
$\mathbf{R}_{E1} \leftarrow \mathbf{E}$	Computation of the $\mathbf{R}$ matrix from the $\mathbf{QR}$ decomposition of E	`xgeqrf`
$\mathbf{u} \leftarrow \mathbf{R}_{E1}^T \backslash \mathbf{c}$	Solving	`xtrsm`
$\mathbf{v} \leftarrow \mathbf{R}_{E1} \backslash \mathbf{u}$	Three triangular	
$\mathbf{p}_{\text{opt}} \leftarrow \mathbf{R}_1 \backslash \mathbf{v}$	Linear systems	

4 Experimental evaluation

4.1 Experimental parameters

In this section, we show the experimental evaluation of the parallel graphic equalizer on the four cores of the Cortex-A57 processor. In the experiments, we have analyzed the effects of using from one to four cores both in terms of execution time and energy consumption of the algorithm. We have evaluated the effect of using the 14 different frequencies allowed by the A57 processors, which vary from 102 to 1479 MHz.

We have performed experiments using different buffer sizes varying from 64 to 8192 samples. One of the goals of our evaluation is to assess the effect of the buffer size in the maximum number of channels that can be processed in real time. In addition, the experiments have been carried out with two main objectives. The first objective was to find the combination of number of cores and frequencies that maximizes the number of channels that can be processed in real time without taking into account the power consumed by the device. The second objective was to take into account the power consumption and find the optimal combination of number of cores and frequencies that allows to process the maximum number of channels per watt in real time.

4.2 Execution time

First, we analyze the evolution of the execution time of the algorithm with the number of channels and the effect of parameters such as the buffer sample size and the number of CPU cores. We run our first experiments at maximum CPU frequency (1479 MHz) and we use an audio buffer size of 4096 samples, which implies a real-time threshold of 92.8 ms at the standard audio sampling frequency of 44.1 kHz.

Figure 3 shows that the execution time increases linearly with the number of channels and that it can be substantially reduced by using more cores. As a consequence, by increasing the number of cores we can also increase the number of channels that can be processed in real time. For example, if we execute the sequential version of the algorithm in one core we can process up to seven channels in real time. However, if we execute the parallel version using the four cores of the CPU, we can process up to 27 channels in real time.

To analyze the advantage of using more CPU cores we show in Fig. 4 the speed-up obtained in the same experiments shown in Fig. 3. If we take as optimum a speed-up equal to the number of cores, then we obtain the optimal speed-up when using two and three cores and near the optimal when using four cores. The saw-tooth pattern shown by the lines is due to the load balancing effect on the different cores. That is, we obtain the best speed-up when we process the same number of channels in every core. Logically, when we increase the total number of channels to process, the load imbalance is reduced and so is the height of the teeth.
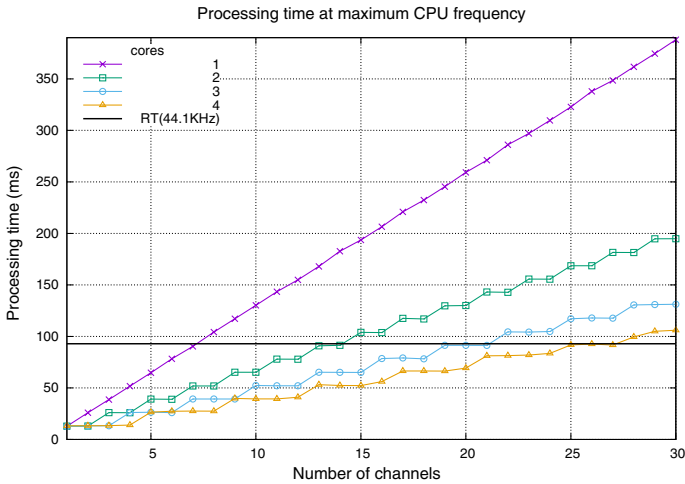
Processing time at maximum CPU frequency

Fig. 3 Processing time of a different number of channels at maximum CPU frequency with a buffer size of 4096 samples
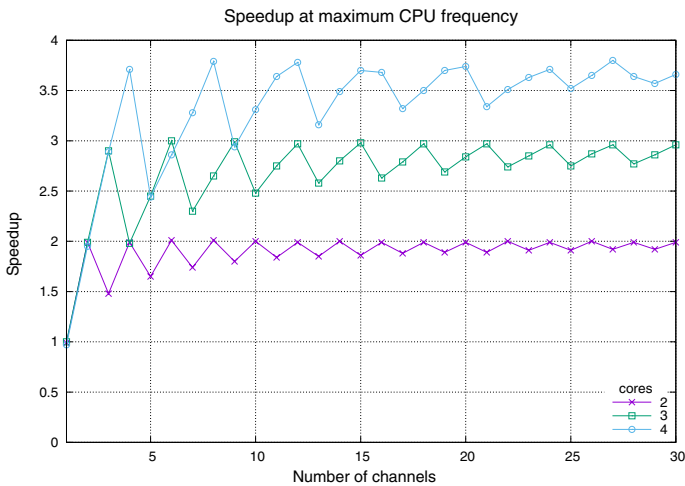
Speedup at maximum CPU frequency

Fig. 4 Speed-up when processing 4 channels at maximum CPU frequency with a buffer size of 4096 samples

Next we analyze the effect of increasing the buffer size on the maximum number of channels that can be processed in real time. Figure 5 shows that the number of channels increases at a steep rate for small buffer sizes, but tends to converge to a maximum for larger sizes of the buffer. This can be explained by the fact that the computational load is made up of two factors: first, the computation of the filter coefficients, which is done only once for each buffer, and second, the filtering process, which is applied sample by sample, and thus depends on the buffer size linearly. Therefore at small buffer sizes, where the coefficient calculation
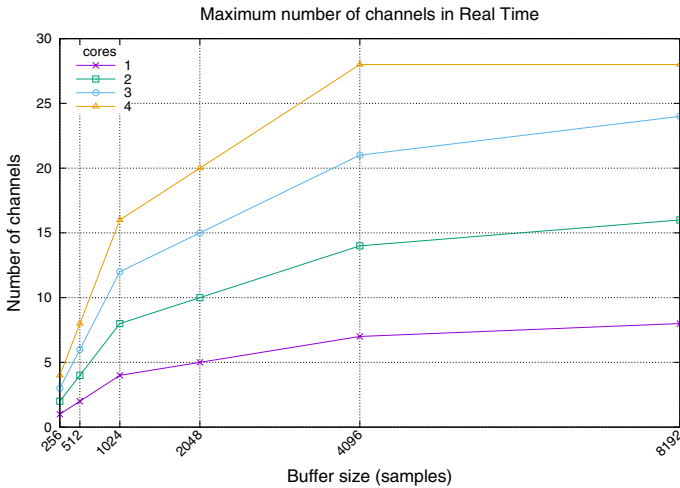
Maximum number of channels in Real Time

Fig. 5 Effect of the buffer size on the maximum number of channels that can be processed in real time

dominates the computational load, increasing the buffer size decreases the average load significantly, thus increases the number of channels we may process. In contrast, at larger buffer sizes the computational complexity gets dominated by the filtering process, and thus there is no further advantage of increasing the buffer size. The disadvantage of too large of a buffer size is larger latency in the processing and constraining to user to update the filter parameters at a smaller rate. Since we approach the maximum throughput with a buffer size of 4096 samples, we use this size in the rest of the experiments. Note that another way of increasing the computational efficiency would be simply not calculating the filter coefficients for each buffer, but only for every tenth buffer, for example, while using smaller buffer sizes. However, this has not been investigated in the present study.

4.3 Energy consumption

One of the main goals of using SoC, such as the one included in the Jetson Nano board, is to combine high computational performance with low energy consumption. In this section, we will evaluate the effect of modifying the frequency of the CPU cores on the number of channels that can be processed in real time, and also on the energy consumption of the platform.

We obtain the energy consumption as the product of the time taken by our algorithm to complete and the measured power dissipation of the whole platform. As we want to measure only the energy consumed by our algorithm, we previously subtract the power dissipated by the operating system processes while running at the default frequency of the CPU.

Firstly, we will evaluate the effect of reducing the frequency of the cores in the processing time of eight channels using a buffer of 4096 samples. Figure 6 shows
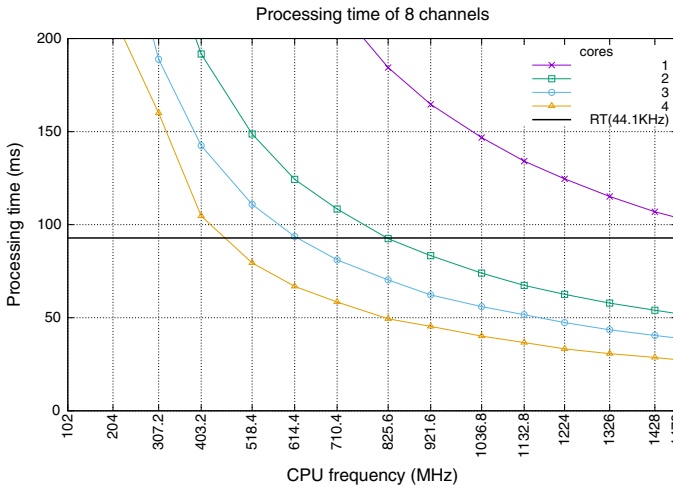
Fig. 6 Effect of the CPU frequency in the processing time of 8 channels

that using only one core we cannot process eight channels in real time even using the maximum frequency. However, if we use two cores, we can reach this number using a wide range of the frequencies of the CPU. As we increase the number of cores, we can process 8 channels in real time with very low frequencies which, as we will see, can greatly reduce the energy consumption of the algorithm.

Figure 7 shows how the maximum number of channels that we can process in real time evolves both with the number of cores and the CPU frequencies. Increasing any
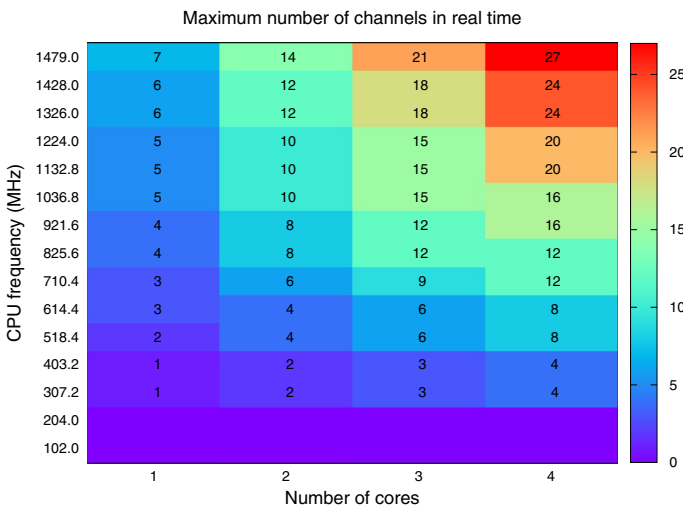

Fig. 7 Effect of the CPU frequency in the maximum number of channels that can be processed in real time. Values are only shown when at least one channel can be processed in real time

of these two parameters increases that number from only one channel using one core at 307.2 MHz to 27 channels using four cores at maximum frequency. The figure also shows that using the two lowest frequencies of the CPU we cannot process any channel in real time even using the four available cores.

Next we show how the CPU frequency and the number of cores affects the energy consumed by the algorithm. Figure 8 shows that as we increase the number of cores, we reduce not only the processing time, but also the energy consumed by the algorithm to process eight channels. Regarding the CPU frequency, increasing it does not always reduce the energy consumption. As a matter of fact, the lowest consumption is obtained using four cores at 1132.8 MHz. Moreover, using frequencies as low as 307.2 MHz with four cores we approach the most efficient case in terms of energy consumption.

On the other hand, if we want to increase the number of channels processed in real time, we have to increase the frequency or the number of cores. However, we pay a price in terms of energy consumption, as it can be seen in Fig. 9.

Finally, in order to assess the energy efficiency of the algorithm we will use a parameter that combines the computational performance and the energy consumption. On many occasions, we may be interested in reducing the time required to process each channel, but without an excessive increase in power consumption. For this purpose, we are interested in finding the value where the energy consumption of processing each channel is minimal. Moreover, in our case we keep the constraint that the channels must be processed in real time. To calculate this value, we can divide the maximum number of channels processed in real time that we can see in Fig. 7 by the power dissipated by the platform during their processing. In this way, we will obtain an efficiency metric that will give us the number of channels per watt in real time. Figure 10 shows again that the maximum energy efficiency is obtained
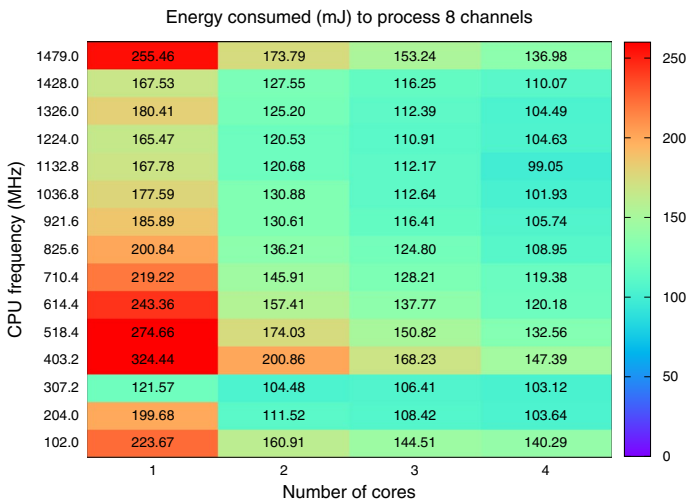

Fig. 8 Energy consumption to process 8 channels varying CPU frequency and number of cores. Buffer size: 4096 samples
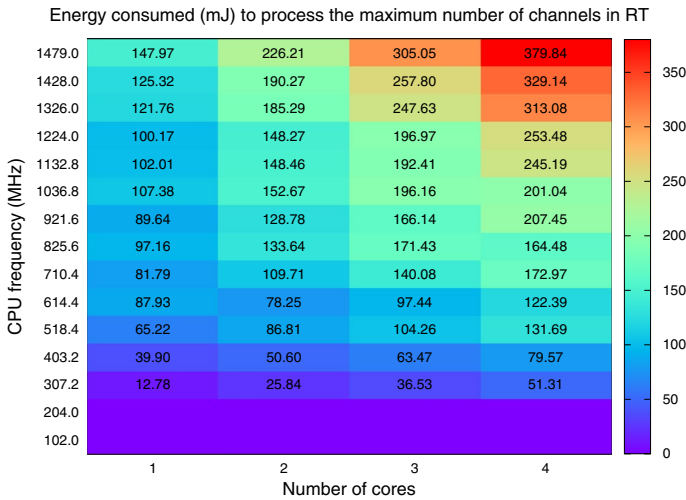
Energy consumed (mJ) to process the maximum number of channels in RT

CPU frequency (MHz)	1	2	3	4
1479.0	147.97	226.21	305.05	379.84
1428.0	125.32	190.27	257.80	329.14
1326.0	121.76	185.29	247.63	313.08
1224.0	100.17	148.27	196.97	253.48
1132.8	102.01	148.46	192.41	245.19
1036.8	107.38	152.67	196.16	201.04
921.6	89.64	128.78	166.14	207.45
825.6	97.16	133.64	171.43	164.48
710.4	81.79	109.71	140.08	172.97
614.4	87.93	78.25	97.44	122.39
518.4	65.22	86.81	104.26	131.69
403.2	39.90	50.60	63.47	79.57
307.2	12.78	25.84	36.53	51.31
204.0				
102.0				

Number of cores

Fig. 9 Energy consumption to process the maximum number of channels in real time for every combination of CPU frequency and number of cores. Values are only shown when at least one channel can be processed in real time
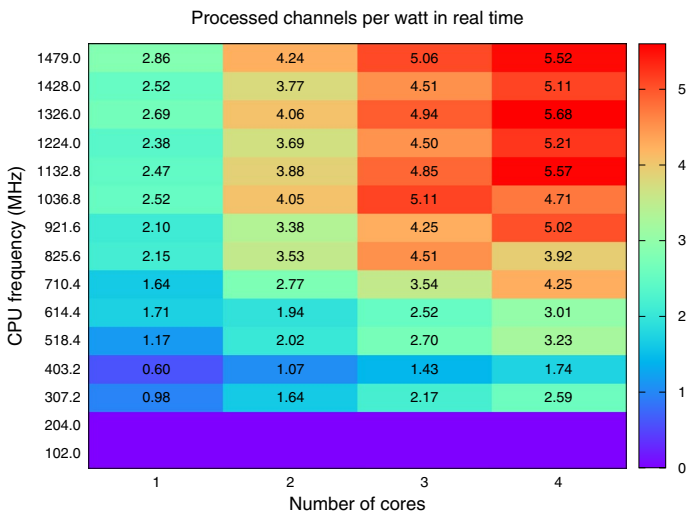
Processed channels per watt in real time

CPU frequency (MHz)	1	2	3	4
1479.0	2.86	4.24	5.06	5.52
1428.0	2.52	3.77	4.51	5.11
1326.0	2.69	4.06	4.94	5.68
1224.0	2.38	3.69	4.50	5.21
1132.8	2.47	3.88	4.85	5.57
1036.8	2.52	4.05	5.11	4.71
921.6	2.10	3.38	4.25	5.02
825.6	2.15	3.53	4.51	3.92
710.4	1.64	2.77	3.54	4.25
614.4	1.71	1.94	2.52	3.01
518.4	1.17	2.02	2.70	3.23
403.2	0.60	1.07	1.43	1.74
307.2	0.98	1.64	2.17	2.59
204.0				
102.0				

Number of cores

Fig. 10 Effect of the CPU frequency and number of cores in the number of channels per watt that can be processed in real time

when using four cores at high frequencies. Specifically, we can process up to 5.68 channels per watt with four cores at 1326 MHz. However, even using three cores we can process more than five channels per watt at several frequencies. As we reduce the frequency and the number of cores, the algorithm is less efficient in terms of channels per watt.

5 Conclusions

In this work, we have analyzed the performance of a multichannel parallel graphic equalizer on an embedded multicore system from audio, computational performance and energy efficiency perspectives. Specifically, we have used the quad-core ARM Cortex-A57 that is embedded on the NVIDIA Jetson Nano board for our experiments. We have been aiming for a configuration that provides an efficient trade-off between computational performance and energy consumption. To this end, we have analyzed different configurations varying the number of cores, the buffer size of the audio samples, as well as the frequency of the CPU cores. From the results, we can extract that it is not necessary to exploit all the cores or to work at the maximum frequency in order to achieve an efficient implementation. Our experiments indicate that using a high CPU frequency and three or four cores, our parallel algorithm is able to equalize more than five channels per watt in real time using an audio buffer of 4096 samples, which implies a latency of 92.8 ms at the standard sample rate of 44.1 kHz.

References

1. Liu Z, Dickson K, McCanny JV (2005) Application-specific instruction set processor for soc implementation of modern signal processing algorithms. IEEE Trans Circuits Syst I Regul Pap 52(4):755–765
2. Andreopoulos Y, Jiang D, Demosthenous A (2010) Prediction-based incremental refinement for binomially-factorized discrete wavelet transforms. IEEE Trans Signal Process 58(8):4441–4447
3. Ren S, Deligiannis N, Andreopoulos Y, Islam MA, van der Schaar M (2014) Dynamic scheduling for energy minimization in delay-sensitive stream mining. IEEE Trans Signal Process 62(20):5439–5448

4. Keller Y, Coifman RR, Lafon S, Zucker SW (2010) Audio-visual group recognition using diffusion maps. IEEE Trans Signal Process 58(1):403–413
5. Rumsey F (2020) Modular synths and embedded computing. J Audio Eng Soc 68(3):234–237
6. "Equalizer at Spotify," https://support.spotify.com/us/article/equalizer/, (accessed 2021 November 17)
7. Ramos G, López JJ (2006) Filter design method for loudspeaker equalization based on IIR parametric filters. J Audio Eng Soc 54(12):1162–1178
8. Rämö J, Välimäki V (2012) Digital augmented reality headset. J Electr Comput Eng. https://doi.org/10.1155/2012/457374
9. Kamaris G, Zachos P, Mourjopoulos J (2021) Low filter order digital equalization for mobile device earphones. J Audio Eng Soc 69(5):297–308
10. Mäkivirta A, Antsalo P, Karjalainen M, Välimäki V (2003) Modal equalization of louspeaker-room responses at low frequencies. J Audio Eng Soc 51(5):324–343
11. Brännmark L-J, Bahne A, Ahlén A (2013) Compensation of loudspeaker-room responses in a robust MIMO control framework. IEEE Trans Audio Speech Lang Process 21(6):1201–1216
12. Välimäki V, Reiss JD (2016) All about audio equalization: solutions and frontiers. Appl Sci 6(5):129
13. Välimäki V, Liski J (2017) Accurate cascade graphic equalizer. IEEE Signal Process Lett 24(2):176–180
14. Rämö J, Välimäki V, Bank B (2014) High-precision parallel graphic equalizer. IEEE/ACM Trans Audio Speech Lang Process 22(12):1894–1904
15. Chen W (1996) Performance of cascade and parallel IIR filters. J Audio Eng Soc 44(3):148–158
16. Belloch JA, Gonzalez A, Martínez-Zaldívar FJ, Vidal AM (2011) Real-time massive convolution for audio applications on GPU. J Supercomput 58(3):449–457
17. Savioja L, Välimäki V, Smith JO (2011) Audio signal processing using graphics processing units. J Audio Eng Soc 59(1–2):3–19
18. Belloch JA, Bank B, Savioja L, Gonzalez A, Välimäki V (2014) Multi-channel IIR filtering of audio signals using a GPU. In: Proceedings of the IEEE International Conference on Acoust, Speech and Signal Processing (ICASSP), Florence, Italy
19. Bank B, Belloch JA, Välimäki V (2017) Efficient design of a parallel graphic equalizer. J Audio Eng Soc 65(10):817–825
20. NVIDIA Corp (2020) NVIDIA jetson linux developer guide. PR-06076-R32
21. Belloch JA, Bank B, Larios D, Igual F, Quintana-Orti E, Vidal AM (2017) Solving weighted least squares (WLS) problems on ARM-based architectures. J Supercomput 73(1):530–542
22. Perez Gonzales E, Reiss J (2009) Automatic equalization of multi-channel audio using cross-adaptive methods. In: Proceedings of the AES 127th Conv. New York
23. Rämö J, Välimäki V (2013) Live sound equalization and attenuation with a headset. In: Proceedings of the AES 51st International Conference, Helsinki, Finland, Aug. (2013)
24. Holters M, Zölzer U (2006) Graphic equalizer design using higher-order recursive filters. In: Proceedings of the Montreal, QC, Sep, Int. Conf. Digital Audio Effects, pp 37–40
25. Tassart S (2013) Graphical equalization using interpolated filter banks. J Audio Eng Soc 61(5):263–279
26. Chen Z, Geng GS, Yin FL, Hao J (2014) A pre-distortion based design method for digital audio graphic equalizer. Digit Signal Process 25:296–302
27. Bank B (2013) Audio equalization with fixed-pole parallel filters: An efficient alternative to complex smoothing. J Audio Eng Soc 61(1/2):39–49
28. Parks TW, Burrus CS (1987) Digit Filter Des. Wiley, USA
29. Bank B (2011) Logarithmic frequency scale parallel filter design with complex and magnitude-only specifications. IEEE Signal Process Lett 18(2):138–141
30. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. Comput Sci Eng, IEEE 5(1):46–55
31. Tomov S, Dongarra J, Baboulin M (2008) Towards dense linear algebra for hybrid GPU accelerated manycore systems. LAPACK Working Note, Tech Rep 210. [Online]. http://www.netlib.org/lapack/lawnspdf/lawn210.pdf
32. Dongarra J, Croz JD, Hammarling S, Hanson RJ (1985) A proposal for an extended set of Fortran basic linear algebra subprograms. ACM Signum Newslett 20(1):2–18

Authors and Affiliations

Jose A. Belloch[1] · **José M Badía**[2] · **German León**[2] · **Balázs Bank**[3] · **Vesa Välimäki**[4]

José M Badía
badia@uji.es

German León
leon@uji.es

Balázs Bank
bank@mit.bme.hu

Vesa Välimäki
vesa.valimaki@aalto.fi

[1]   Depto. de Tecnología Electrónica, Universidad Carlos III de Madrid, Leganés, Spain

[2]   Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I de Castelló, Castellón de la Plana, Spain

[3]   Department of Measurements and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

[4]   Acoustics Lab, Department of Signal Processing and Acoustics, Aalto University, Espoo, Finland