OPEN ACCESS Freely available online

J. A. Belloch, J. M. Badia, G. Leon, and V. Välimäki,
"Efficient Velvet-Noise Convolution in Multicore Processors,"
*J. Audio Eng. Soc.*, vol. 72, no. 6, pp. 383–393 (2024 Jun.).
https://doi.org/10.17743/jaes.2022.0156.

# Efficient Velvet-Noise Convolution in Multicore Processors

**JOSE A. BELLOCH,**[1, *] **JOSE M. BADIA,**[2] **GERMAN LEON,**[2] **AND VESA VÄLIMÄKI,**[3] *AES Fellow*
(jbelloc@ing.uc3m.es) (badia@uji.es) (leon@uji.es) (vesa.valimaki@aalto.fi)

[1]*Department of Electronic Technology, Universidad Carlos III de Madrid, E-28911 Leganés, Spain*
[2]*Department of Computer Science and Engineering, Universitat Jaume I, E-12071 Castellón de la Plana, Spain*
[3]*Acoustics Lab, Department of Information and Communications Engineering, Aalto University, FI-02150 Espoo, Finland*

Velvet noise, a sparse pseudo-random signal, finds valuable applications in audio engineering, such as artificial reverberation, decorrelation filtering, and sound synthesis. These applications rely on convolution operations whose computational requirements depend on the length, sparsity, and bit resolution of the velvet-noise sequence used as filter coefficients. Given the inherent sparsity of velvet noise and its occasional restriction to a few distinct values, significant computational savings can be achieved by designing convolution algorithms that exploit these unique properties. This paper shows that an algorithm called the transposed double-vector filter is the most efficient way of convolving velvet noise with an audio signal. This method optimizes access patterns to take advantage of the processor's fast caches. The sequential sparse algorithm is shown to be always faster than the dense one, and the speedup is linearly dependent on sparsity. The paper also explores the potential for further speedup on multicore platforms through parallelism and evaluate the impact of data encoding, including 16-bit and 32-bit integers and 32-bit floating-point representations. The results show that using the fastest implementation of a long velvet-noise filter, it is possible to process more than 40 channels of audio in real time using the quad-core processor of a modern system-on-chip.

## 0 INTRODUCTION

Velvet noise is a special type of sparse ternary pseudorandom sequence discovered by Karjalainen and Järveläinen [1]. The basic velvet-noise signal contains only the sample values $-1$, 0, and 1. The main use of velvet noise in audio engineering has been reverberation algorithms based on feedback [1–4] and feedforward structures [1, 5–8]. Another application of velvet noise is the decorrelation of audio signals [9–12], a process that allows reducing the correlation of signals [13, 14] and, for example, distributing processed copies of a mono signal to multiple loudspeakers to produce a diffuse sound field [15, 16]. Velvet noise has also been used in sound [17–20] and speech synthesis [21, 22].

Many velvet-noise applications require discrete convolution, or filtering a digital audio signal with a finite impulse response (FIR) filter, which becomes expensive when multiple filters are applied in parallel, such as in source spreading, or on the outputs of a large multichannel system [13, 23]. The fast convolution method is sometimes used to reduce the computational load by implementing the convolution as a multiplication of spectra in the frequency domain [24, 25]. The efficiency then comes from the use of the fast Fourier transform but leads to inevitable latency due to block processing [24, 25]. However, the fact that the velvet-noise sequence is composed of a large number of non-zero values implies that the corresponding multiplications and additions can be skipped during filtering. This has led the authors to look for a new method that aims to exploit the memory hierarchy of current processors, including fast caches, as well as the parallelism provided by their multicore architectures [26].

The use of multicore processors has made it possible to meet the demand for computing resources [27] in a variety of different applications, including audio processing [28]. There are currently low-power, multicore system-on-chip platforms that allow high levels of audio processing throughput to be achieved in automotive systems, as shown

*To whom correspondence should be addressed, email: jbelloc@ing.uc3m.es. Last updated: March 15, 2024 by Beckmann et al. [29]. The present authors' previous study used multicore processors to implement a multichannel parallel graphic equalizer that is efficient in terms of both computational performance and power consumption [30].

The NVIDIA Jetson Nano is a commercial off-the-shelf multicore platform that has become quite popular because it houses the Tegra X1 chip, which is used in a wide range of mobile devices such as the Nintendo Switch game console [31] or in drone systems for object recognition [32]. The Tegra X1 embedded chip is based on a quad-core ARM Cortex-A57 CPU running at 1.43 GHz with 4 GiB of LPDDR4 memory and a 128-core NVIDIA Maxwell GPU. One of its key features is its low power consumption, combined with multiple levels of parallelism, resulting in high performance per watt.

Until now, the literature related to velvet-noise processing [33, 5] showed their results by using a straightforward FIR processing based on performing the operations without considering their values. The present authors' contribution is to implement a high-performance sequential and parallel version of this process. The sequential version uses several techniques to improve the performance of the filtering. First, it exploits the large sparsity of the filter to greatly reduce the number of computations. Second, it sorts the computations to reduce the number of memory accesses and take advantage of the processor's fast caches.

One of the major limitations to achieving high performance is the slowness of memory access compared to the speed of computation. Properly organizing data access to hide memory latency is critical to increasing the performance of algorithms. A key factor is to reuse data once it has been loaded from memory and load it from the fastest cache memories in the architecture. This is realized using well-known techniques such as reordering and unrolling loops or block processing of data [34; 35, Chap. 6]. A practical example of the use of such techniques is the implementation of the product of matrices in a numerical algebra library called BLAS [36]. In this paper, these techniques are applied to improve the performance of the audio filtering algorithm.

This paper tests firstly the different proposed implementations on different sample resolutions, including 16-bit and 32-bit integers, and a 32-bit floating-point encoding. Finally, OpenMP [37] is used to implement a parallel version of the algorithm that can exploit the multicore architecture of the CPU to process a multi-channel audio source. The authors evaluate how the number of cores running in parallel affects the computational performance of a multi-channel velvet-noise–based application of varying lengths. The Tegra X1 chip embedded in the NVIDIA Jetson Nano platform is used.

This paper is structured as follows. SEC. 1 describes the velvet-noise signal and its use in FIR filtering of audio signals. SEC. 2 discusses different sparse FIR filter implementation techniques. SEC. 3 explores the performance of the different implementations in terms of the maximum number of channels that can be convolved in real time. Finally, SEC. 4 closes the paper with concluding remarks.

## 1 FIR FILTERING WITH VELVET NOISE

Velvet noise is a sparse sequence in which a single positive or a negative impulse appears on a fixed time interval [1]. The impulse locations can be determined as follows:

$$k(m) = \text{round}[mT_\text{d} + r_1(m)(T_\text{d} - 1)], \tag{1}$$

where $m = 0, 1, 2, ...$ is the impulse index, $T_\text{d}$ is the range of samples where only a single impulse is allowed, and $r_1(m)$ is a random number drawn from a uniform distribution (0,1) [38]. In the original velvet noise, the sign of each impulse is also chosen using random numbers, so that only two non-zero signal values are possible: +1 or $-1$ [1, 38]. The signs can be easily produced by rounding and scaling the random numbers. The rest of the samples in the velvet-noise sequence are zero, and this is why the sequence is called sparse.

In practice, velvet noise sounds like smooth, featureless white noise, when the impulse density is 2,000 impulses per second or larger [38]. When the sample rate is 44.1 kHz, this corresponds to an impulse for every $T_\text{d} = 22$ samples, which means that only 4.5% of the samples are non-zero. This original velvet noise has a flat power spectrum [39] and sounds smoother than Gaussian random noise [1, 38]. It has been learned that in special cases, much denser velvet noise can sound fine, for example, when the velvet noise is low-pass filtered [33, 5, 40]. These features, together with its sparsity, suggest that velvet noise can be used to efficiently filter signals with noise.

In practice, the samples of the velvet-noise signal are often assigned as the coefficients of an FIR filter. The filtering is realized by convolving the input audio signal $x(n)$ with the FIR filter coefficients $s(n)$ to obtain the output signal $y(n)$:

$$y(n) = s(n) * x(n) = \sum_{j=0}^{L-1} s(j)x(n-j), \tag{2}$$

where the asterisk (*) denotes the discrete convolution and $L$ is the number of FIR filter coefficients, also called the filter length.

The computational savings in velvet-noise convolution come from the observation that many of the coefficients in $s(n)$ are zero, and as the non-zero values appear at predetermined locations, it is possible to skip the multiplications and additions associated with the zeros. This is why a direct implementation of Eq. (2), which also realizes the multiplications of signal samples by zero coefficients, may be highly wasteful. The economic version of the velvet-noise convolution, which skips the zeros, can be written as

$$y(n) = x(n) * s(n) = \sum_{m=0}^{M-1} s[k(m)]x[n - k(m)], \tag{3}$$

where vector $k$ stores the indices of the non-zero values in $s$, and so, the index $m$ runs through the $M$ non-zero impulses contained in $s[k(m)]$. Thus, only the multiplications and additions involving the non-zero coefficients are executed. Since a small portion, for example, 5%, of the values of $s(n)$ are non-zero, i.e., $M << L$, the implementation using Eq.

(3) can be more efficient than the direct implementation of Eq. (2).

An additional observation is that, when the original velvet noise is used, the non-zero coefficients can only have values −1 and 1, which do not require a multiplication but rather an addition or subtraction. Thus, in a further simplification of Eq. (3), each impulse is directly associated with an addition or subtraction based on its sign [33, 5]:

$$y(n) = \sum_j x[n - k_+(j)] - \sum_m x[n - k_-(m)], \tag{4}$$

where $k_+$ and $k_-$ are arrays containing the indices of the positive and negative impulse location within the velvet-noise sequence, respectively. In practice then, Eq. (4) first adds the input signal values coinciding with the positive impulses, then adds the input signal values coinciding with the negative impulses, and finally subtracts the two sums. This implementation of the velvet-noise convolution is possible in practice, when the filter is time-invariant, i.e., the coefficient values are not changed during run time.

In various audio applications, velvet noise is used in a modified form so that it can contain floating-point values, not just values ±1. This is necessary, for example, in the decorrelation application in which an exponentially decaying impulse response is preferred [9, 10]. For this reason, this study also considers the FIR filtering with a sparse sequence of arbitrary coefficient values. In this case, Eq. (3) must be used to perform the convolution.

## 2 SPARSE DESIGN FOR VELVET-NOISE FILTERING

The sparse design is based on reducing the number of computations and also on sorting them out so that the number of memory accesses can be reduced. This allows the fast cache memory of the architecture can be used [26].

### 2.1 Practical Example

The best way to understand the authors' high-performance sequential implementation of the velvet-noise processing is to use a small example. Straightforwardly following Eq. (3) and giving example values to the indices in $s$ of $M = 4$ non-zero impulses, $k(m) = \{1, 3, 6, 7\}$, **X** can be defined as a vector of size $N$ that contains the input audio samples to be filtered with the velvet-noise sequence, **Y** as a vector that contains the filtered output samples, **S** as a vector of size $L$ that holds the filter coefficient values of the velvet-processing (zero and non-zero values), and $x_i$, $y_j$, and $s_k$ as the elements $i$, $j$, and $k$ from vectors **X**, **Y**, and **S**, respectively. Note that the first sample index in vectors **X** and **Y** is one. Therefore, vectors **X** and **Y** start with the samples $x_1$ and $y_1$, respectively. Fig. 1(a) shows the block diagram of the example sparse FIR filter in direct form.

The FIR processing can be split into three main processing steps. **Step 1** corresponds to the computation of the first output $L$ samples and, in this example, involves the following operations:

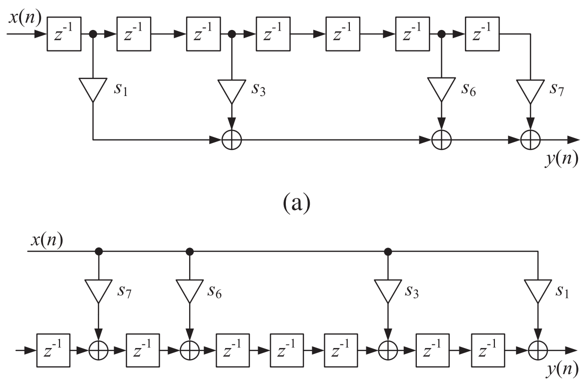


Fig. 1. Sparse FIR filter structure with non-zero coefficients $s_1$, $s_3$, $s_6$, and $s_7$ in (a) direct form and (b) transpose form. Notice the reverse order of the coefficients and delays in (b).

$$\begin{aligned}
y_1 &= 0\\
y_2 &= x_1 \cdot s_1\\
y_3 &= x_2 \cdot s_1\\
y_4 &= x_3 \cdot s_1 + x_1 \cdot s_3\\
y_5 &= x_4 \cdot s_1 + x_2 \cdot s_3\\
y_6 &= x_5 \cdot s_1 + x_3 \cdot s_3\\
y_7 &= x_6 \cdot s_1 + x_4 \cdot s_3 + x_1 \cdot s_6
\end{aligned}$$

Color/gray tones at the equations are made to discriminate each one of the contributions of each non-zero coefficient value to the output sample. Note that the size $L$ corresponds in the example $L = 7$ since seven is the last index in vector **S**. **Step 2** corresponds to the computation of the rest of the samples except the last $L$ samples and involves the following operations:

$$\begin{aligned}
y_8 &= x_7 \cdot s_1 + x_5 \cdot s_3 + x_2 \cdot s_6 + x_1 \cdot s_7\\
y_9 &= x_8 \cdot s_1 + x_6 \cdot s_3 + x_3 \cdot s_6 + x_2 \cdot s_7\\
y_{10} &= x_9 \cdot s_1 + x_7 \cdot s_3 + x_4 \cdot s_6 + x_3 \cdot s_7\\
y_{11} &= x_{10} \cdot s_1 + x_8 \cdot s_3 + x_5 \cdot s_6 + x_4 \cdot s_7\\
y_{12} &= x_{11} \cdot s_1 + x_9 \cdot s_3 + x_6 \cdot s_6 + x_5 \cdot s_7\\
\cdots &= \cdots + \cdots + \cdots\\
y_N &= x_{N-1} \cdot s_1 + x_{N-3} \cdot s_3 + x_{N-6} \cdot s_6 + x_{N-7} \cdot s_7\\
y_{N+1} &= x_N \cdot s_1 + x_{N-2} \cdot s_3 + x_{N-5} \cdot s_6 + x_{N-6} \cdot s_7
\end{aligned}$$

Finally, **Step 3** corresponds to the computation of the last six samples that go from the output sample $y_{N+2}$ to the sample $y_{N+7}$, as indicated here:

$$\begin{aligned}
y_{N+2} &= x_{N-1} \cdot s_3 + x_{N-4} \cdot s_6 + x_{N-5} \cdot s_7\\
y_{N+3} &= x_N \cdot s_3 + x_{N-3} \cdot s_6 + x_{N-4} \cdot s_7\\
y_{N+4} &= x_{N-2} \cdot s_6 + x_{N-3} \cdot s_7\\
y_{N+5} &= x_{N-1} \cdot s_6 + x_{N-2} \cdot s_7\\
y_{N+6} &= x_N \cdot s_6 + x_{N-1} \cdot s_7\\
y_{N+7} &= x_N \cdot s_7
\end{aligned}$$

Considering the operations carried out in the described three steps, the authors can extract that **Step 2** holds the main part of the total processing, which increases also with the size $N$, and shows a regular structure. In this sense, the authors focus their efforts on carrying out an efficient implementation of the operations shown in **Step 2**. To this end, they need to reduce not only the number of operations but also the number of memory accesses and try to exploit the fast cache memory of the architecture. Each memory access can be orders of magnitude more expensive than each addition or multiplication performed by the algorithm [26]. The memory accesses must be sorted out so that each element loaded is reused as much as possible before storing the result. Besides, whenever possible, the authors must try to access consecutive elements of the different vectors so that they are stored in neighboring memory locations and, thus, they are loaded in the fast cache memory.

## 2.2 Algorithm Implementations

This section proposes five versions of the algorithm using different strategies to access the elements of the vectors and perform the computations.[1] Next, the authors show the pseudo-code of the different versions used to compute the group of $N - L + 1$ filtered audio samples that correspond to the operations executed in **Step 2**. In all cases, computations are performed only using the $M$ non-zero elements of the velvet-noise filter applied to the corresponding input samples.

In the basic version, shown in Listing 1, the computation of each filtered output sample is completed before starting the next one. To avoid the multiplication on each iteration of the internal loop, a conditional `if` sentence is used to add the input sample whenever the filter coefficient is +1 and to subtract the input sample whenever the coefficient is −1. This algorithm is called the Conditional Filter (COF).

Listing 1. COF.

```
for each element y[i] of Y
  for each element k[m] of k
    if ( k[m] == 1)
      y[i] = y[i] + x[i − k[m]]
    else / ( k[m] == −1 )
      y[i] = y[i] − x[i − k[m]]
```

In some cases, evaluating the `if` clause included in the COF version may be more expensive than performing the product of the filter coefficient with the sample value. To test this, the authors propose another version of the algorithm shown in Listing 2, which they call the sparse multiplier filter (SMF) and corresponds to Eq. (3).

Listing 2. SMF.

```
for each element y[i] of Y
  for each element k[m] of k
    y[i] = y[i] + s[k[m]] * x[i − k[m]]
```

Both the multiplications and the conditional sentence can be avoided when the indices of the positive and negative filter coefficients are stored in two different vectors, $k_+$ and $k_-$, as suggested in Eq. (4). Listing 3 shows this double-vector filter (DVF) version of the algorithm.

Listing 3. DVF.

```
for each element y[i] of Y
  for each element $k_+$[m] of $k_+$
    y[i] = y[i] + x[i − $k_+$[m]]
  for each element $k_-$[m] of $k_-$
    y[i] = y[i] − x[i − $k_-$[m]]
```

In the first three versions of the algorithm, consecutive elements of vector **Y** are always accessed so that in most cases they are stored in the fast cache memory. Moreover, as the same element is updated in successive iterations of the innermost loop, the computations can be performed in a register, and the result is stored in the memory location of $y[i]$ at the end of each iteration of the outermost loop. On the contrary, the authors are accessing non-consecutive elements of vectors **X** and **S** on each iteration of the innermost loops, which does not employ the cache memory.

To reduce the memory access cost, the order of the two loops of the code can be reversed. This corresponds to the transpose FIR filter structure [41], which is shown for the example FIR filter of SEC. 2.1 in Fig. 1(b). In this structure, the current input sample $x(n)$ is multiplied by all filter coefficients and added to the appropriate locations of the output buffer **Y**. The idea of transposing a digital filter structure, which is used in Fig. 1(b), is sometimes called flow graph reversal and has been known for decades in signal processing [42, pp. 204–205].

Listing 4 shows the transpose form of the algorithm, where the same strategy as in version 3 is used to avoid the products. In this case, each filter coefficient is added (or subtracted) to all the filtered samples before using the next filter coefficient. If referring to the operations associated with **Step 2** of the method shown in SEC. 2, the authors add (or subtract) first the samples shown in black color, then the samples shown in red color, and finally the samples shown in blue color. As can be seen, the authors are always accessing consecutive elements of both the **X** and **Y** vectors, which greatly reduces the memory access cost, as intended. Structurally, this corresponds to the transposed, or backward, version of the DVF algorithm, and it is therefore called the transposed double-vector filter (DVF Transpose), see Listing 4.

Listing 4. DVF Transpose.

```
for each element $k_+$[m] of $k_+$
  for each element y[i] of Y
    y[i] = y[i] + x[i − $k_+$[m]]
for each element $k_-$[m] of $k_-$
  for each element y[i] of Y
    y[i] = y[i] − x[i − $k_-$[m]]
```

The authors can try to reduce the memory access cost even more by avoiding accessing the elements of the vectors **X** and **Y** twice, first to apply the positive and then the negative signs. To remove these accesses, the authors again

[1] https://github.com/josembadia/velvet_audio.

Table 1. Computational cost and memory access cost of the five velvet-noise FIR filtering algorithms. Note that the authors discriminate between consecutive (**con**) and non-consecutive (non-con) accesses, i.e., whether vector elements appear in neighboring memory locations when they are loaded or stored. The last two algorithms try to avoid non-consecutive memory accesses.

| | | Memory accesses to the main vectors | | | |
|---|---|---|---|---|---|
| Algorithm version | Computational cost | Y | X | S | k |
| **1. COF** | *LM* | ***L* con** | *LM* non-con | 0 | *LM* non-con |
| **2. SMF** | 2*LM* | ***L* con** | *LM* non-con | *LM* non-con | *LM* non-con |
| **3. DVF** | *LM* | ***L* con** | *LM* non-con | 0 | *LM* non-con |
| **4. DVF Transpose** | *LM* | ***LM* con** | ***LM* con** | 0 | ***L* con** |
| **5. SMF Transpose** | 2*LM* | ***LM* con** | ***LM* con** | *M* non-con | ***M* con** |

have to pay the price of multiplications by −1 and 1, as shown in Listing 5. For each iteration of the outermost loop, the same filter coefficient $s[k[m]]$ is always used in all iterations of the innermost loop. This means that is enough to load this element from memory once per iteration of the outermost loop, further reducing the memory access cost. This algorithm version is the transpose of the SMF algorithms (Listing 2), and the authors therefore call it the transposed sparse multiplier filter (SMF Transpose).

Listing 5. SMF Transpose.

```
for each element k[m] of k
    for each element y[i] of Y
        y[i] = y[i] + s[k[m]] * x[i − k[m]]
```

Versions 1, 3, and 4 have been designed assuming that the filter coefficients only have values −1, 0, and 1. However, versions 2 and 5 can also be used in the general case, where the filter coefficients can have any real value.

Table 1 summarizes the computational cost in terms of additions and multiplications of the different versions of the algorithm. It also includes the number of accesses of the different algorithm versions to the main vectors used, differentiating between accesses to consecutive and non-consecutive elements, which has a large effect on the memory access cost. Taking into account only the theoretical costs shown in Table 1, the fastest algorithm would be the DVF Transpose, because it has a low computational cost and it always accesses consecutive elements of every main vector. On the other hand, Table 1 suggests that the first three algorithms, COF, SMF, and DVF, may lead to a slow implementation, because they require a large number of non-consecutive memory accesses. In the next section, the authors experimentally compare the different algorithm versions and analyze the influence of memory access costs on the execution time.

## 3 EXPERIMENTAL EVALUATION

This section presents studies on the execution time of the different algorithm implementations for a short and long FIR filter. The maximum number of channels that can be convolved in real time is also determined for each algorithm version.

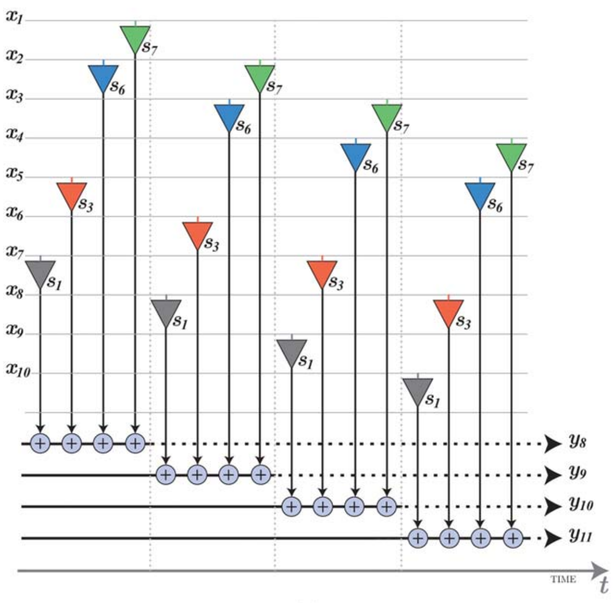
### 3.1 Experimental Environment

All the experiments were performed in an NVIDIA Jetson Nano Development Kit containing a low-power Tegra X1 (TX1) System on Chip (SoC) [43]. This SoC is manufactured in 20-nm planar technology and contains a quad-core ARM Cortex-A57 CPU implementing the ARMv8-A 64-bit architecture. Each core has a 48-KiB L1 instruction cache, 32-KiB L1 data cache, and 2-MiB L2 unified cache shared by all cores. The SoC also includes an NVIDIA Maxwell GPU with 128 CUDA cores. It includes a 256-KiB L2 cache, 64-KiB shared memory, and 64-K (32-bit) registers. Both the CPU and GPU share 4 GiB of external DDR4 memory.

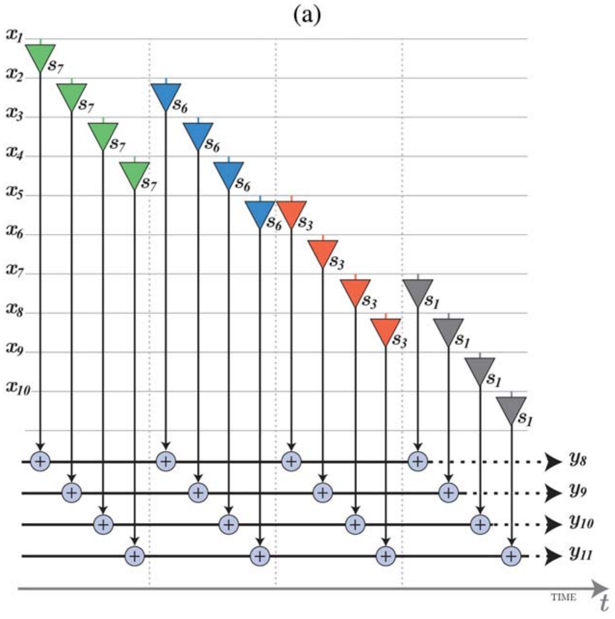
The code was implemented in C language and parallelized using OpenMP. The authors chose the OpenMP framework to perform the tests because of its widespread recognition and portability in parallel programming environments. OpenMP is a parallel programming interface originally designed for shared memory multiprocessors. It operates on a fork-join execution model, where tasks are divided among multiple threads at different points in the code and later joined together. Programming with OpenMP involves using a set of functions and inserting directives at specific points in the code. It inherently supports data parallelism, allowing different threads to run on separate processor cores, working on the same tasks but with different data. In addition, OpenMP facilitates task parallelism, allowing each thread to perform different tasks.

### 3.2 Results: Single-Channel Filtering

Evaluations were performed with two filters: a short filter with a size of $L = 1{,}320$ taps and a large filter with a size of $L = 88{,}000$ taps. These tests also considered real-time processing, which means that the input audio samples were processed block by block using three different block sizes. Fig. 2 shows an example of the block versions of the direct and transposed forms of the FIR forms of the filter shown



(a)



(b)

Fig. 2. Block versions of the sparse FIR filter with blocksize of 4 and non-zero coefficients $s_1$, $s_3$, $s_6$, and $s_7$ in the (a) the direct form (SMF) and (b) the transpose form (SMF Transpose), cf. Fig. 1.

in Fig. 1. Input samples are processed from top to bottom, while blocks of outputs are processed from left to right.

Fig. 2 uses the same example described in SEC. 2.1, so the filter length is $L = 7$ with $M = 4$ non-zero coefficients, namely $s_1$, $s_3$, $s_6$, and $s_7$. In this case, *blocksize* $= 4$. In the direct-form implementation shown in Fig. 2(a), all the operations to produce each output $y_i$ are completed before starting the operations corresponding to the next one, $y_{i+1}$. On the contrary, in the transposed-form implementation of Fig. 2(b), each non-zero coefficient is applied to each output of the block before the next coefficient is applied. Therefore, the outputs are necessarily generated by blocks. Specifically, the computational performance of these algorithms is tested using block sizes of 1,024; 2,048; and 4,096 audio samples, for a sample rate of $f_s = 44.1$ kHz. This means, for example, that the application runs in real time as long as the processing time of a 1,024-sample block is less than 23.22 ms ($blocksize/f_s$).

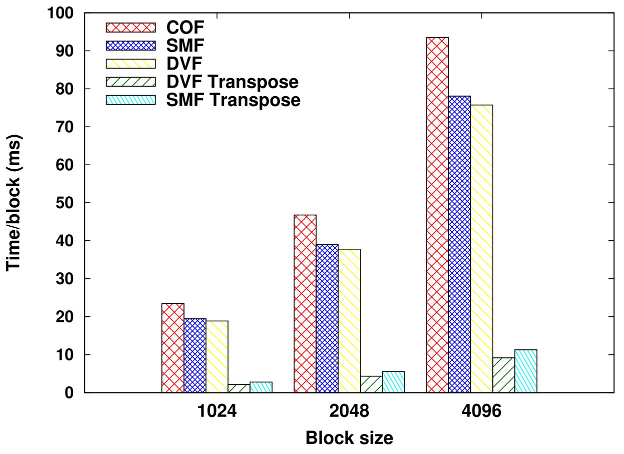


Fig. 3. Execution time per block of the different versions of the sparse FIR filtering algorithm using 32-bit integers. The filter length is $L = 88{,}000$ coefficients, out of which $M = 4{,}000$ are non-zero.

First, the authors evaluate the efficiency of the five versions of the algorithm described in SEC. 2. The different versions are run using the large filter ($L = 88{,}000$ coefficients) and $M = 4{,}000$ non-zero elements ($T_d = 22$) distributed with values $\{+1, -1\}$, i.e., the filter has a sparsity of 4.5%. All computations were performed using 32-bit integers, and the authors obtained the execution times with the three block sizes.

Fig. 3 shows the execution time per block in milliseconds for the different versions of the code while processing one audio channel. The results show that the last two versions of the algorithm are much faster than the first three. Version 4 of the algorithm is almost nine times faster than version 3. This means that the main reduction in execution time is due to the swapping of the two main loops of the algorithm. Changing the order in which the elements of the different vectors are accessed allows accessing elements in neighboring memory locations, thus making better use of the fast cache of the processor. This reduction in memory access time implies a large reduction in the execution time of the algorithm. Other modifications to the code have a much smaller effect on the execution times of the different versions of the algorithm. For example, version 2 is slightly faster than version 1, which means that performing the product by the filter coefficient at each iteration is faster than evaluating a conditional sentence to avoid it.

In fact, versions 2 and 3 and versions 4 and 5 have similar execution times, so their relative behavior may change in different experimental environments. For example, version 5 may be faster than version 4 if a different processor with different memory or different processing speeds is used. Also, their behavior may change depending on the compiler or compilation options. To further examine this, the same speed test was repeated in another multicore processor, the

Table 2. Execution times of the sparse and dense algorithms per block for the long FIR filter with different data types and block sizes. Only the fastest sparse algorithm v4 (DVF Transpose) is considered here.

| | | Execution time (ms) | | |
|---|---|---|---|---|
| Block size | Algorithm | int16 | int32 | float |
| 1,024 | Sparse (v4) | 1.38 | 2.23 | 2.64 |
| | Dense | 27.2 | 53.28 | 46.94 |
| 2,048 | Sparse (v4) | 2.15 | 4.56 | 4.88 |
| | Dense | 53.46 | 107.93 | 97.71 |
| 4,096 | Sparse (v4) | 4.93 | 9.64 | 10.58 |
| | Dense | 108.39 | 224.769 | 217.88 |

Table 3. Comparison of the time per block of the sparse DVF Transpose (v4) and the dense algorithms to apply the short filter with three different data types and a block size of 1,024 samples.

| | Execution time (ms) | | |
|---|---|---|---|
| Algorithm | int16 | int32 | float |
| Sparse (v4) | 0.02 | 0.04 | 0.04 |
| Dense | 0.36 | 0.71 | 0.63 |

Intel Core i7-10700. The order of the execution times of the algorithm versions was the same as in Fig. 3, but, whereas version 4 was around 11 times faster than version 1 in the ARM processor, it was more than 106 times faster in the Intel processor. This is due to the larger cache memories of the Intel architecture. This result confirms the advantage gained by swapping loops and making better use of this type of fast memory.

In the rest of the experiments, the fastest version of the algorithm, which is version 4, is always used. To demonstrate the benefits of exploiting filter sparsity, a "dense" variant of the method has been implemented, using a filter that includes both zero and non-zero elements. The dense filter is compared with the fastest sparse variant (v4) described in SEC. 2, which only stores and applies the non-zero elements to the samples in two vectors. The dense variant uses standard dot vector products to multiply blocks of samples by the vector containing the filter coefficients, including both zero and non-zero values.

Table 2 shows the execution time of both variants of the method with different block sizes and element data types. In particular, the sparse variant of the method proves to be about 20 times faster than its dense counterpart in all scenarios. The choice of data type also has an impact on performance. In particular, the use of 16-bit integers results in a two-fold increase in speed compared to 32-bit integers, while the use of floats (32-bit single-precision floating-point elements) only marginally outperforms the 32-bit integer option. In addition, using longer filters allows better exploitation of filter sparsity. Table 3 shows that by using a significantly shorter filter with the same percentage of non-zero elements, the sparse variant of the method is between 15 and 18 times faster than its dense counterpart depending on the data type.

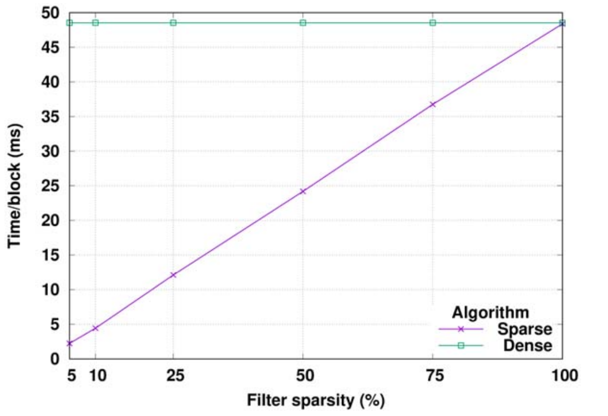


Fig. 4. Effect of the sparsity on the execution time of the sparse and dense FIR filtering algorithms (block size = 1,024 samples).

To demonstrate the optimal utilization of filter sparsity by their sparse method, the authors conducted a comparison of its execution time with the dense variant using filters of varying sparsity levels. Six different FIR filters of length $L$ = 88,000 were generated, which have a different sparsity between 5% and 100% and, thus, a different number of non-zero elements. The non-zero coefficients were generated randomly. Fig. 4 visually depicts the linear increase in execution time for the sparse version as the sparsity of the filter is decreased. Notably, the execution time becomes equivalent to that of the dense version when the filter contains 100% non-zero elements.

## 3.3 Results: Multi-Channel Filtering

To test the multicore resources of this processor, the proposed sparse convolution is applied to multiple audio channels, which serves as a prime example of embarrassingly parallel code. Each channel can undergo the same convolution process in parallel, allowing for efficient workload distribution and achieving nearly optimal parallel performance. When an equal number of channels is distributed among cores, a balanced workload is attained. Fig. 5 demonstrates the linear increase in execution time as the number of channels increases. However, by increasing the number of cores utilized for parallel convolutions, a significant improvement in the real-time processing of channels can be achieved. While the sequential version of the code can process ten channels in real time (see the topmost line in Fig. 5), employing the parallel version with four available cores (the lowest line in Fig. 5) allows for the real-time processing of up to 40 channels.

Finally, Fig. 6 provides valuable insights into the advantages of leveraging filter sparsity and parallelizing the processing of multiple audio channels. The results clearly demonstrate that even when running the dense variant of the code in parallel with four cores, it falls short of processing even one channel in real time. Although the parallel version gives a speed improvement over its sequential counterpart, the execution time is still twice the real-time threshold,

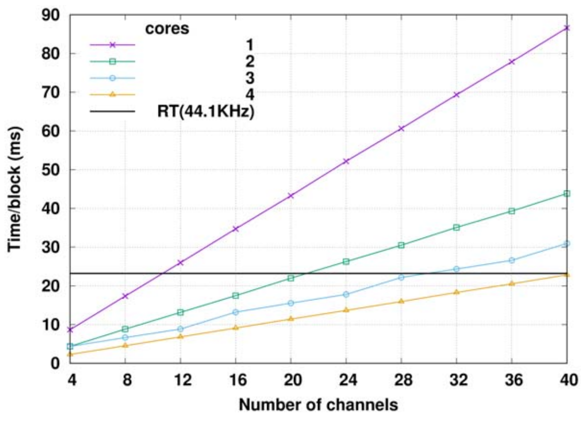


Fig. 5. Execution time of the parallel version of the sparse algorithm with the large FIR filter using a block size of 1,024 samples. The horizontal line shows the real-time threshold of 23.2 ms for the sample rate of 44.1 kHz.

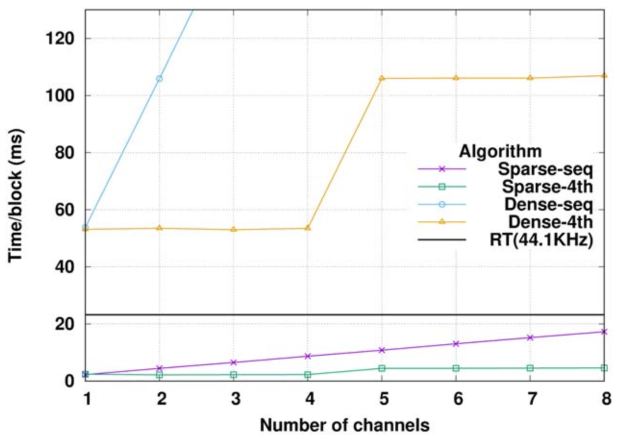


Fig. 6. Comparison of the execution time of the sequential and parallel algorithms with the large velvet-noise FIR filter using a block size of 1,024 samples.

even when processing between one and four channels. In contrast, the sequential sparse implementation enables real-time filtering of up to ten channels. Moreover, when employing the parallel sparse version, the number of channels that can be processed scales nearly linearly with the number of cores employed.

## 4 CONCLUSION

This paper introduces a novel sparse convolution method, called the DVF Transpose, designed to significantly enhance the computational efficiency of audio applications relying on velvet noise. These explorations have delved into various strategies, with a primary focus on optimizing the sequence in which elements within different vectors are accessed. Experiments with a multicore system-on-chip reported in this paper revealed a breakthrough: accessing elements located in close memory proximity results in an almost nine-fold acceleration of the FIR filtering process. Specifically, the DVF Transpose version of the algorithm is 8.77 times faster than the DVF-only version because of inverting the order of the two main loops and accessing consecutive elements in memory.

The results also show that the sequential sparse version is always faster than a conventional dense FIR filter. The speedup is linearly proportional to the sparsity of the filter. Furthermore, this paper examined the capabilities of the multicore processor, demonstrating that it can seamlessly handle more than 40 velvet-noise–based FIR filters in a real-time application.

Additionally, this study looked into the impact of data encoding and resolution on processing. The authors found that both floating-point elements (32-bit single-precision floating-point values) and 32-bit integers exhibit comparable processing speeds, providing the flexibility to employ real numbers as coefficients in such applications. This versatility enhances the adaptability of velvet-noise–based audio systems.

## 5 ACKNOWLEDGMENT

This work was started during a 1-month research visit of the first author to the Aalto Acoustics Lab in August 2022. The research visit was funded by "Ayuda Movilidad Programa Propio de Investigación 2022, modalidad A: jóvenes doctores, de la Universidad Carlos III de Madrid." The work has also been supported by the Spanish Ministry of Science and Innovation under projects PID2022-137048OA-C43, PID2019-106455GB-C21, PID2020-113656RB-C21, and TED2021-131401A-C22 funded by MCIN/AEI/10.13039/501100011033 and by the Regional Government of Madrid through the project MIMACUHSPACE-CM-UC3M. This work has been supported by the Madrid Government (Comunidad de Madrid-Spain) under the Multiannual Agreement with UC3M in the line of Excellence of University Professors (EPUC3M26), and in the context of the V PRICIT (Regional Programme of Research and Technological Innovation). This research was part of the activities of the Nordic Sound and Music Computing Network (NordForsk project no. 86892).

## 6 REFERENCES

[1] M. Karjalainen and H. Järveläinen, "Reverberation Modeling Using Velvet Noise," in *Proceedings of the AES 30th International Conference on Intelligent Audio Environments*, pp. 1–7 (Saariselkä, Finland) (2007 Mar.).

[2] K.-S. Lee, J. S. Abel, V. Välimäki, T. Stilson, and D. P. Berners, "The Switched Convolution Reverberator," *J. Audio Eng. Soc.*, vol. 60, no. 4, pp. 227–236 (2012 Apr.).

[3] S. J. Schlecht and E. A. P. Habets, "Scattering in Feedback Delay Networks," *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 28, pp. 1915–1924 (2020 Jun.). https://doi.org/10.1109/TASLP.2020.3001395.

[4] V. Välimäki and K. Prawda, "Late-Reverberation Synthesis Using Interleaved Velvet-Noise Sequences," *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 29, pp. 1149–1160 (2021 Feb.). https://doi.org/10.1109/TASLP.2021.3060165.

[5] V. Välimäki, B. Holm-Rasmussen, B. Alary, and H.-M. Lehtonen, “Late Reverberation Synthesis Using Filtered Velvet Noise,” *Appl. Sci.*, vol. 7, no. 5, paper 483 (2017 May). https://doi.org/10.3390/app7050483.

[6] J. Fagerström, S. J. Schlecht, and V. Välimäki, “Dark Velvet Noise,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 192–199 (Vienna, Austria) (2022 Sep.).

[7] S. Lee, H.-S. Choi, and K. Lee, “Differentiable Artificial Reverberation,” *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 30, pp. 2541–2556 (2022 Jul.). https://doi.org/10.1109/TASLP.2022.3193298.

[8] S. Riedel, M. Frank, and F. Zotter, “The Effect of Temporal and Directional Density on Listener Envelopment,” *J. Audio Eng. Soc.*, vol. 71, no. 7/8, pp. 455–467 (2023 Jul./Aug.). https://doi.org/10.17743/jaes.2022.0088.

[9] B. Alary, A. Politis, and V. Välimäki, “Velvet-Noise Decorrelator,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 405–411 (Edinburgh, UK) (2017 Sep.).

[10] S. J. Schlecht, B. Alary, V. Välimäki, and E. A. P. Habets, “Optimized Velvet-Noise Decorrelator,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 87–94 (Aveiro, Portugal) (2018 Sep.).

[11] K. Prawda, S. J. Schlecht, and V. Välimäki, “Multichannel Interleaved Velvet Noise,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 208–215 (Vienna, Austria) (2022 Sep.).

[12] S. J. Schlecht, J. Fagerström, and V. Välimäki, “Decorrelation in Feedback Delay Networks,” *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 31, pp. 3478–3487 (2023 Sep.). https://doi.org/10.1109/TASLP.2023.3313440.

[13] G. S. Kendall, “The Decorrelation of Audio Signals and Its Impact on Spatial Imagery,” *Comput. Music J.*, vol. 19, no. 4, pp. 71–87 (1995 Winter). https://doi.org/10.2307/3680992.

[14] C. Anemüller, A. Adami, and J. Herre, “Efficient Binaural Rendering of Spatially Extended Sound Sources,” *J. Audio Eng. Soc.*, vol. 71, no. 5, pp. 281–292 (2023 May). https://doi.org/10.17743/jaes.2022.0069.

[15] G. Potard and I. Burnett, “Decorrelation Techniques for the Rendering of Apparent Sound Source Width in 3D Audio Displays,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 280–284 (Naples, Italy) (2004 Oct.).

[16] V. Pulkki and J. Merimaa, “Spatial Impulse Response Rendering II: Reproduction of Diffuse Sound and Listening Tests,” *J. Audio Eng. Soc.*, vol. 54, no. 1/2, pp. 3–20 (2006 Jan./Feb.).

[17] S. D’Angelo and L. Gabrielli, “Efficient Signal Extrapolation by Granulation and Convolution With Velvet Noise,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 107–112 (Aveiro, Portugal) (2018 Sep.).

[18] V. Välimäki, J. Rämö, and F. Esqueda, “Creating Endless Sounds,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 219–226 (Aveiro, Portugal) (2018 Sep.).

[19] K. J. Werner, “Generalizations of Velvet Noise and Their Use in 1-Bit Music,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, paper 53 (Birmingham, UK) (2019 Sep.).

[20] J. Fagerström, S. J. Schlecht, and V. Välimäki, “One-to-Many Conversion for Percussive Samples,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 129–135 (Vienna, Austria) (2021 Sep.).

[21] H. Kawahara, K.-I. Sakakibara, M. Morise, et al., “Frequency Domain Variants of Velvet Noise and Their Application to Speech Processing and Synthesis,” in *Proceedings of INTERSPEECH*, pp. 2027–2031 (Hyderabad, India) (2018 Sep.).

[22] T. Uchida and M. Morise, “A Practical Method of Generating Whisper Voice: Development of Phantom Silhouette Method and Its Improvement,” *Acoust. Sci. Tech.*, vol. 42, no. 4, pp. 214–217 (2021 Jul.). https://doi.org/10.1250/ast.42.214.

[23] V. Välimäki, J. D. Parker, L. Savioja, J. O. Smith, and J. S. Abel, “Fifty Years of Artificial Reverberation,” *IEEE Trans. Audio Speech Lang. Process.*, vol. 20, no. 5, pp. 1421–1448 (2012 Jul.). https://doi.org/10.1109/TASL.2012.2189567.

[24] F. Wefers and M. Vorländer, “Optimal Filter Partitions for Real-Time FIR Filtering Using Uniformly-Partitioned FFT-Based Convolution in the Frequency-Domain,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 155–161 (Paris, France) (2011 Sep.).

[25] M. Vorländer, “Convolution and Binaural Sound Synthesis,” in *Auralization: Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality*, RWTHedition, pp. 135–144 (Springer, Cham, Switzerland, 2020). https://doi.org/10.1007/978-3-030-51202-6_9.

[26] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface, ARM Edition* (Morgan Kaufmann, Cambridge, MA, 2016).

[27] G. Andrade, D. Griebler, R. Santos, and L. G. Fernandes, “A Parallel Programming Assessment for Stream Processing Applications on Multi-Core Systems,” *Comput. Stand. Interfaces*, vol. 84, paper 103691 (2023 Mar.). https://doi.org/10.1016/j.csi.2022.103691.

[28] M. García-Valls, “Integrating Multicore Awareness Functions Into Distribution Middleware for Improving Performance of Distributed Audio Surveillance,” *Adv. Eng. Softw.*, vol. 132, pp. 92–100 (2019 Jun.). https://doi.org/10.1016/j.advengsoft.2019.01.003.

[29] P. Beckmann, J. Whitecar, J. Peil, and K. Hegde, “Achieving Maximum Audio Processing Throughput on the Latest Chipsets,” in *Proceedings of the AES International Conference on Automotive Audio* (2022 Jun.), paper 21. http://www.aes.org/e-lib/browse.cfm?elib=21822.

[30] J. A. Belloch, J. M. Badia, G. Leon, B. Bank, and V. Välimäki, “Multicore Implementation of a Multichannel Parallel Graphic Equalizer,” *J. Supercomput.*, vol. 78, pp. 15715–15729 (2022 Sep.). https://doi.org/10.1007/s11227-022-04495-3.

[31] A. van de Velde, “Nintendo Switch Uses Stock NVIDIA Tegra X1 T210 CPU & GM20B Maxwell Core,” https://wccftech.com/nintendo-switch-tegra-x-1-nvidia-maxwell (accessed Aug. 25, 2022).

[32] P. A. Rad, D. Hofmann, S. A. Pertuz Mendez, and D. Goehringer, “Optimized Deep Learning Object Recognition for Drones Using Embedded GPU,” in *Proceedings of the 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–7 (Vasteras, Sweden) (2021 Sep.). https://doi.org/10.1109/ETFA45728.2021.9613590.

[33] B. Holm-Rasmussen, H.-M. Lehtonen, and V. Välimäki, “A New Reverberator Based on Variable Sparsity Convolution,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 344–350 (Maynooth, Ireland) (2013 Sep.).

[34] M. D. Lam, E. E. Rothberg, and M. E. Wolf, “The Cache Performance and Optimizations of Blocked Algorithms,” *ACM SIGOPS Operat. Syst. Rev.*, vol. 26, pp. 63–74 (1991 Apr.). https://doi.org/10.1145/106973.106981.

[35] R. E. Bryant and D. R. O’Hallaron, *Computer Systems: A Programmer’s Perspective* (Prentice Hall, Boston, MA, 2011).

[36] L. S. Blackford, A. Petitet, R. Pozo, et al., “An Updated Set of Basic Linear Algebra Subprograms (BLAS),” *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151 (2002 Jun.). https://doi.org/10.1145/567806.567807.

[37] OpenMP, *OpenMP Application Programming Interface v. 5.2*, OpenMP Architecture Review Board (2021 Nov.).

[38] V. Välimäki, H.-M. Lehtonen, and M. Takanen, “A Perceptual Study on Velvet Noise and Its Variants at Different Pulse Densities,” *IEEE Trans. Audio Speech Lang. Process.*, vol. 21, no. 7, pp. 1481–1488 (2013 Jul.). https://doi.org/10.1109/TASL.2013.2255281.

[39] N. Meyer-Kahlen, S. J. Schlecht, and V. Välimäki, “Colours of Velvet Noise,” *Electron. Lett.*, vol. 58, no. 12, pp. 495–497 (2022 Jun.). https://doi.org/10.1049/ell2.12501.

[40] J. Roberts, J. Fagerström, S. J. Schlecht, and V. Välimäki, “How Smooth Do You Think I Am: An Analysis on the Frequency-Dependent Temporal Roughness of Velvet-Noise,” in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, pp. 312–318 (Copenhagen, Denmark) (2023 Sep.).

[41] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing* (Pearson, Harlow, Essex, UK, 2014), 3rd ed.

[42] J. O. Smith, *Introduction to Digital Filters With Audio Applications* (W3K Publishing, 2007). http://www.w3k.org/books/.

[43] NVIDIA, *NVIDIA Tegra X1: NVIDIA’S New Mobile Superchip*, White Paper, v1.0 (2015 Jan.).

## THE AUTHORS

Jose A. Belloch

Jose M. Badia

German Leon

Vesa Välimäki

Jose A. Belloch received a degree in telecommunications engineering, Master's degree in parallel and distributed computing, and Ph.D. degree in computer science from the Universitat Politècnica de València, Valencia, Spain, in 2007, 2010, and 2014, respectively. Since 2017, he has been an Assistant Professor at Universidad Carlos III de Madrid, Madrid, Spain. He collaborates closely with the Acoustics Lab, Aalto University, Espoo, Finland. He carried out an internship with the Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary. His work is focused on applying new parallel architectures to signal processing algorithms. He has developed several real-time audio applications related to multichannel massive filtering, binaural sound, wave field synthesis systems, and sound source localization using general-purpose graphics processing units and ARM architectures.

•

Jose M. Badia was born in Valencia, Spain. He received his B.S. degree in Computer Science from the Polytechnic University of Valencia, Spain, in 1991 and obtained his Ph.D. degree in Computer Science in 1996. Since 1994, he has been a member of the Department of Computer Science and Engineering of the University Jaume I of Castellón, Spain, where he was Teaching Assistant from 1994 to 1997, Assistant Professor to 2000, and Associate Professor since 2000. From 2007 to 2013, he was head of this department. He has published more than 50 papers in international conferences and journals. His main research interests include high-performance computing, energy-aware computing, digital audio processing, system-on-chip reliability, and quantum computing.

•

German Leon was born in Valencia, Spain. He received B.S. and Ph.D. degrees in Computer Science from the Polytechnic University of Valencia, Spain. He is a member of the Department of Computer Science and Engineering of the University Jaume I of Castellón, Spain. His pre-doctoral research lines focused on the design of digital systems in the field of robotics and quality of service (QoS) networks, as well as on circuit synthesis methodology for Field-Programmable Gate Arrays (FPGAs) based on high-level descriptions. The postdoctoral research line has been framed in the High Performance Computing & Architectures (HPCA) research group, which has consisted of addressing the optimization and consumption-aware computing of numerical algorithms for general purpose multicore processors as well as accelerators (GPU, DSP, and FPGA).

•

Vesa Välimäki is Professor of audio signal processing and Vice Dean for research at Aalto University, Espoo, Finland. He received his M.Sc. and D.Sc. degrees from the Helsinki University of Technology in 1992 and 1995, respectively. In 1996, he was a Postdoctoral Research Fellow at the University of Westminster, London, UK. In 2001–2002, he was a Professor of signal processing at the Pori unit of the Tampere University of Technology. In 2008–2009, he was a visiting scholar at the Stanford University Center for Computer Research in Music and Acoustics. He is a Fellow of the AES, IEEE, and Asia-Pacific Artificial Intelligence Association. He was the General Chair of the 11th International Conference on Digital Audio Effects (DAFx) in 2008 and the 14th International Sound and Music Computing Conference (SMC) in 2017. Prof. Välimäki is the Editor-in-Chief of the *Journal of the Audio Engineering Society.*