EXPLORING PARALLELISM AND ENERGY EFFICIENCY IN A MULTISTAGE LINEAR-PHASE OCTAVE FILTER BANK

Jose M. Badia
Dept. of Computer Science and Engineering
Universitat Jaume I
E-12071 Castellón de la Plana, Spain
badia@uji.es

Jose A. Belloch
Dept. of Electronic Technology
Universidad Carlos III de Madrid
E-28911 Leganés, Spain
jbelloc@ing.uc3m.es

Vesa Välimäki
Acoustics Lab
Dept. of Information and Communications Engineering
Aalto University
FI-02150 Espoo, Finland
vesa.valimaki@aalto.fi

ABSTRACT

This paper presents a high-performance and energy-aware implementation of a multistage linear-phase octave filter bank for edge system-on-chip (SoC) platforms. The algorithm relies on a cascade of stretched FIR filter stages and complementary band splitting to preserve linear phase across all outputs. While effective, mapping such structures to embedded multicore CPUs introduces significant challenges regarding state management, task synchronization, memory-traffic efficiency, and energy-aware execution. These issues are especially relevant in block-based edge-audio processing, where high throughput must be balanced against the power constraints of mobile and embedded devices. We derive a cache-friendly sequential realization using a blocked streaming schedule and compact circular state. Building on this, we propose a parallel design based on an OpenMP task pipeline with explicit dependencies to preserve the filter-bank semantics without fine-grained synchronization in the filtering tasks. Experimental results on an NVIDIA Jetson Orin Nano module show that the optimized sequential version already sustains more than 1.18 M samples/s, while the task-level pipeline reaches speedups above $4.5\times$ for suitable block sizes. Furthermore, our analysis reveals a clear trade-off between throughput and power, showing that the most energy-efficient operating point does not necessarily coincide with maximum performance on multicore edge SoCs.

1. INTRODUCTION

Audio processing is increasingly moving towards edge and Internet of Things (IoT) environments, where signals are captured and processed close to the point of use rather than continuously streamed to remote servers [1]. This trend is especially relevant in battery-powered and embedded scenarios, such as mobile devices, smart headphones, hearing-assistance systems, smart speakers, and other local audio-processing platforms [2]. In such contexts, the objective is not only to provide sufficient processing throughput, but also to use the available computational, memory, and power resources efficiently.

Modern commercial off-the-shelf embedded system-on-chip (SoC) devices provide substantially more computing capability than traditional low-power platforms, often integrating multicore CPUs, low-power GPUs, accelerators, and power-management mechanisms. A representative example is the NVIDIA Jetson Orin Nano, which combines a six-core Arm Cortex-A78AE CPU, an Ampere-class embedded GPU, and LPDDR5 memory in a compact edge-oriented platform [3]. In addition, these devices expose configurable operating points, such as CPU/GPU frequency settings and power modes, which make it possible to adapt performance to the needs of the application [4, 5]. Dedicated audio DSPs and heterogeneous audio-processing platforms are also relevant in this domain, as shown by previous work on modern audio chipsets and heterogeneous equalizer implementations [6, 7]. Nevertheless, CPU-based implementations remain relevant because they are accessible through standard C/OpenMP toolchains, portable across a broad range of programmable edge SoCs, and suitable for rapid prototyping, research systems, and applications in which audio processing is integrated with larger CPU-side workloads or the audio DSP is not exposed through an openly programmable interface. Efficient edge-audio processing should therefore be studied not only as a question of arithmetic complexity, but also as a trade-off between programmability, portability, throughput, memory behavior, and energy consumption [1, 8].

This work focuses on the efficient realization of the multistage linear-phase octave filter bank introduced by Bruschi *et al.* [9]. That design is based on the interpolated finite impulse response (IFIR) filtering principle and constructs the octave decomposition from a prototype half-band FIR low-pass filter, its stretched versions, and complementary branches [9]. The resulting structure is attractive for audio equalization because it preserves linear phase while remaining substantially more efficient than straightforward long-FIR realizations and introducing much less latency than FFT-based alternatives [9]. Linear-phase equalization is especially useful in applications where phase distortion is undesirable, such as multichannel processing, parallel audio paths, phase-compatible signal chains, and crossover design [10].

From an audio-effects perspective, this class of filter bank is most useful when octave-band spectral shaping must preserve the phase relationships between recombined signal paths. Typical use cases include phase-coherent graphic equalization for playback or mastering, multichannel and spatial-audio equalization, loudspeaker or headphone correction, and parallel or multiband processing chains in which phase mismatches between bands may introduce coloration [10]. In embedded products such as smart speakers, portable playback devices, or edge audio processors, an energy-aware implementation can help make this type of processing practical under CPU and power constraints. At the same time, the fixed group delay of the fully linear-phase structure makes it

less appropriate for ultra-low-latency monitoring; the target scenarios are therefore those in which this latency can be tolerated, compensated, or traded for phase coherence.

The present work does not modify the filter-bank design itself. Instead, it studies how to realize the existing multistage structure efficiently on a CPU-based edge SoC. The implementation evaluated in this paper is a block-based filtering kernel operating on preallocated buffers. It is intended to characterize throughput, scalability, and energy behavior of the filtering computation, rather than to provide a complete low-latency audio-callback or plugin-host implementation. This distinction is important because callback-level systems introduce additional concerns, such as host scheduling, wake-up costs, idle-state behavior, and deadline enforcement, which require a different experimental setup.

Starting from the IFIR octave equalizer structure, the first step of this work is an efficient sequential implementation. Rather than focusing only on reducing arithmetic operations, the implementation is designed to minimize data movement and memory-access costs, which are often decisive on modern SoCs [6, 11]. To this end, the sequential implementation does not materialize the FIR filters after stretching; instead, it stores each stage in a compact circular state, accumulates directly into the aligned output buffer, and processes the signal in blocks to improve locality. This preserves the filter-bank behavior and provides the baseline for the parallel version.

On top of this optimized baseline, a parallel implementation is developed by exploiting the natural cascade organization of the multistage filter bank. Once the computation is reformulated in terms of sample blocks, the cascade exposes a pipeline form of parallelism in which different blocks can be processed simultaneously at different stages. The proposed parallel version therefore uses task parallelism with OpenMP, where the fundamental unit of concurrent work is the processing of one block at one pipeline stage [12]. The implementation combines stage–block task decomposition, explicit dependency management, sliding-window intermediate buffers, thread-local output accumulation, and reduction tasks in order to preserve the streaming semantics of the original filter while reducing synchronization overhead and maintaining good locality.

Experimental results on the Jetson Orin Nano show that the proposed block-based implementation provides high throughput while revealing a clear throughput–energy trade-off. The optimized sequential implementation sustains more than 1.18 M samples/s, already well above standard audio sampling rates for single-channel processing. The task-pipelined version further reduces the processing time for suitable block sizes, reaching speedups above $4.5\times$ with six CPU threads. At the same time, the minimum observed energy consumption is 0.43 mJ per 2048-sample block at 1036.8 MHz using a single core, showing that the most energy-efficient operating point does not necessarily coincide with the fastest one.

The main contributions of this paper are as follows:

- an efficient sequential realization of the multistage linear-phase octave filter bank, designed to reduce memory traffic and improve data locality while preserving the original filter-bank behavior;
- an OpenMP task-based pipeline parallelization that exploits the cascade structure through stage–block concurrency while preserving the required inter-stage and per-stage state dependencies;
- an experimental evaluation on a commercial embedded edge SoC considering execution time, CPU frequency, dynamic power, and energy consumption; and
- an analysis of the performance–energy trade-off showing that the most suitable operating point depends on the requirements of the target block-based edge-audio workload.

To support reproducibility, the C/OpenMP implementation used in the experiments is publicly available [13].

The remainder of the paper is organized as follows. Section 2 reviews the most relevant related work. Section 3 presents the optimized sequential implementation. Section 4 describes the parallel task-based pipeline. Section 5 reports the experimental evaluation in terms of performance and energy consumption. Finally, Section 6 concludes the paper.

2. RELATED WORK

Linear-phase graphic equalization involves a trade-off between spectral accuracy, latency, and computational cost. Classical FIR realizations preserve phase but often require either high-order filters or FFT-based convolution, both of which may introduce latency or redesign overhead when the target response changes. A key step toward efficient octave-band linear-phase equalization was the IFIR-based structure of Bruschi *et al.* [9], which uses a half-band prototype FIR filter, its stretched versions, and complementary branches to realize the octave bands at much lower complexity than direct long-FIR or FFT-based approaches. Their design reduces latency and arithmetic cost while preserving a linear-phase response. A later hybrid extension replaced the lowest-band FIR section with an IIR shelving filter, further reducing latency at the cost of quasi-linear phase at very low frequencies [14]. The present work keeps the fully linear-phase multistage design and focuses on its efficient block-based realization on an edge multicore CPU.

Previous work on equalizer implementation has explored different forms of parallelism and different execution targets. Norilo *et al.* [7] implemented a low-latency parallel graphic equalizer on a heterogeneous PC–WaveCore platform using Kronos for code generation. Belloch *et al.* [8] studied a multichannel parallel graphic equalizer on an embedded quad-core ARM platform, including throughput–energy trade-offs with OpenMP. These works show the relevance of heterogeneous and multicore execution for audio equalization, although they mainly exploit coarse-grain parallelism across filters, channels, or platform components. In contrast, the filter bank considered here is a cascade of dependent linear-phase stages, which motivates pipeline parallelism across blocks and stages while preserving per-stage state.

Memory hierarchy and execution mapping are also critical in embedded audio processing. In sparse FIR convolution, Belloch *et al.* [11] showed that cache-aware memory access can improve performance on multicore processors. At the platform level, Beckmann *et al.* [6] examined how caching, memory architecture, and multicore configuration affect audio throughput on modern automotive DSPs and SoCs. These results support the view that audio performance on embedded platforms cannot be understood from arithmetic complexity alone; memory locality, data layout, and hardware mapping are often equally important.

Related low-delay filter-bank work has also appeared in adjacent embedded-audio domains. For example, Swamy and Alex [2] proposed a reconfigurable hearing-aid filter bank with a parallel

structure to reduce delay. Although the goals and structures differ from the fully linear-phase octave equalizer considered here, such work further illustrates the importance of matching the filtering structure, implementation strategy, and target platform in embedded audio systems.

3. EFFICIENT SEQUENTIAL IMPLEMENTATION

We follow the multistage linear-phase octave filter bank introduced in [9], whose structure is summarized in Fig. 1. Let the prototype FIR have n_{coef} taps and group delay

$$D = \frac{n_{\mathrm{coef}} - 1}{2}. \tag{1}$$

The filter bank comprises N_f cascaded stages and produces N_f+1 frequency bands. At stage f, the prototype low-pass filter is applied in stretched form with stretch factor $s = 2^f$, corresponding to $H_{\mathrm{LP}}(z^s)$, without explicitly materializing the zero-inserted coefficients.

At each stage, the complementary band, H_{HP}, is obtained by subtracting the low-pass output from a delayed version of the stage input. The stage delay is

$$D_f = 2^f D, \tag{2}$$

so that the complementary contribution can be written as

$$x^{(f)}_{\mathrm{band}}[n] = x^{(f)}[n - D_f] - x^{(f)}_{\mathrm{LP}}[n], \tag{3}$$

where $x^{(f)}$ is the input stream of stage f and $x^{(f)}_{\mathrm{LP}}$ is its low-pass output. The delayed value must be read from the stage history before the current sample is inserted into the state, in order to preserve the streaming semantics of the complementary split.

To time-align all subbands at the final summation node, each stage contribution is accumulated with the synchronization shift

$$\Delta_f = \left(2^{N_f} - 2^{f+1}\right) D. \tag{4}$$

Rather than explicitly delaying intermediate band signals, the implementation shifts the output index to $p = n + \Delta_f$ and accumulates

$$y[p] \mathrel{+}= \left(x_{\mathrm{del}} - x_{\mathrm{LP}}\right) g[N_f - f], \qquad p < L. \tag{5}$$

For the last stage, the residual low-pass term is also added at the same aligned position with gain $g[0]$. This direct aligned accumulation avoids storing full delayed subband signals while preserving the same input-output behavior.

3.1. Block-based Sequential Schedule

Algorithm 1 gives the sequential blocked schedule used as the baseline for the parallel version. The input is processed in blocks of B samples. For each block, the stages are evaluated in cascade order, and two temporary buffers of length B are used in a ping-pong scheme: one stores the current stage input block and the other stores the low-pass output block passed to the next stage. This keeps intermediate data contiguous within a block and defines the same stage–block granularity later used by the task pipeline.

The order of operations inside the innermost loop is important. The delayed value is read before the low-pass update, and the output accumulation follows the aligned rule in (5). Therefore, the blocked traversal changes the execution order and the locality of intermediate data, but not the mathematical behavior of the filter bank.

Algorithm 1 Blocked sequential multistage filter bank

Require: Input $x[0..L-1]$, prototype taps $b[0..n_{\mathrm{coef}}-1]$, gains $g[0..N_f]$, block size B
Ensure: Output $y[0..L-1]$

$D \leftarrow (n_{\mathrm{coef}} - 1)/2$; $\quad y[\cdot] \leftarrow 0$
Allocate two block buffers $\mathtt{A}[0..B-1]$ and $\mathtt{B}[0..B-1]$
Initialize the circular state of each stage $f = 0, \ldots, N_f - 1$
for $i_{\mathrm{start}} = 0$ to $L-1$ step B **do** ▷ Sample blocks
  $B_c \leftarrow \min(B,\ L - i_{\mathrm{start}})$
  $\mathtt{A}[0..B_c-1] \leftarrow x[i_{\mathrm{start}}..i_{\mathrm{start}} + B_c - 1]$
  for $f = 0$ to $N_f - 1$ **do** ▷ Filter stages
    $s \leftarrow 2^f$; $\quad D_f \leftarrow Ds$
    for $j = 0$ to $B_c - 1$ **do** ▷ Samples in a block
      $n \leftarrow i_{\mathrm{start}} + j$
      $x_{\mathrm{del}} \leftarrow \mathrm{ReadDelay}(f, D_f)$
      $x_{\mathrm{LP}} \leftarrow \mathrm{LP\_IFIR}(\mathtt{A}[j], b, s, f)$
      $\mathtt{B}[j] \leftarrow x_{\mathrm{LP}}$
      UpdateOutput$(y, f, N_f, D, n, x_{\mathrm{del}}, x_{\mathrm{LP}}, g, L)$
    end for
    Swap $\mathtt{A} \leftrightarrow \mathtt{B}$
  end for
end for

3.2. Compact IFIR State

The low-pass operation at stage f is evaluated using the IFIR form of the stretched prototype filter,

$$x^{(f)}_{\mathrm{LP}}[n] = \sum_{k=0}^{n_{\mathrm{coef}}-1} b[k]\ x^{(f)}[n - ks], \qquad s = 2^f. \tag{6}$$

Thus, the number of multiply–accumulate operations per output sample is n_{coef}, independently of the stretched support length. The stretch factor only affects which past samples are read.

To implement (6) efficiently, each stage maintains a compact circular state buffer containing the recent samples required by both $\mathrm{ReadDelay}$ and the strided FIR accesses. New samples are inserted by advancing a write pointer modulo the state length, avoiding shifts of the delay line. This reduces the spatial cost from full-length intermediate stage signals to a fixed-size per-stage state, improves cache locality, and preserves the required streaming semantics. The resulting blocked sequential implementation therefore provides a strong locality-aware baseline for the task-parallel pipeline described in the next section.

4. PARALLEL PIPELINE IMPLEMENTATION

The multistage cascade described in Algorithm 1 is particularly well suited to a pipeline parallelization strategy. A key observation is that a block-based streaming schedule (Algorithm 1) already defines a natural unit of work that preserves both streaming semantics and bounded scheduling latency: one stage applied to one block. This granularity enables a task-parallel pipeline where multiple blocks are simultaneously in flight at different stages, while each block advances through the cascade in order. With this decomposition, the pipeline depth equals the number of stages N_f, and the number of blocks determines how well the pipeline remains filled. When the number of blocks is sufficiently large compared to N_f, a long steady-state regime is obtained and high parallel efficiency is expected.

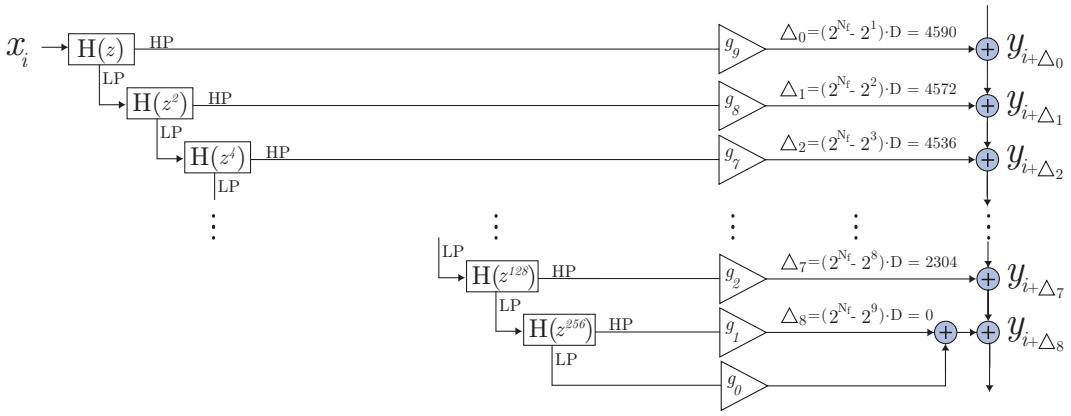

Figure 1: Block diagram of the multistage linear-phase octave filter bank with $N_f = 9$ stages, yielding 10 frequency bands. Each stage f applies a low-pass IFIR structure (`LP`) and forms the complementary high-pass band (`HP`); all subbands are synchronized by the shift Δ_f before the final summation.

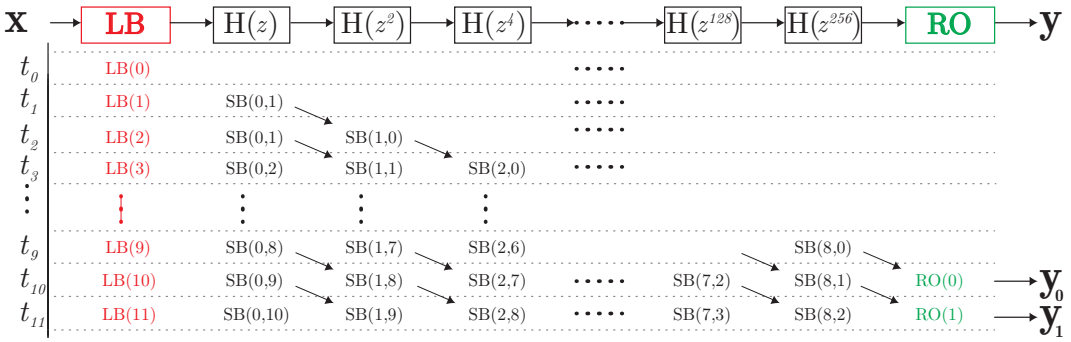

Figure 2: Timing diagram of the initial steps of the task-parallel pipeline for a multistage filter ($N_f = 9$ stages). $LB(i)$ loads the i-th sample block, $SB(j, i)$ executes filter stage j on block i, and $RO(i)$ performs the aligned accumulation of the filtered contributions from block i into the output buffer.

This approach reuses the sequential blocked schedule of Algorithm 1. The same block size B and the same conceptual triple-loop structure (blocks, stages, samples) are retained, but the iteration space is mapped to a set of tasks that can execute concurrently subject to explicit dependencies. Because LP_IFIR can be implemented with the IFIR-style kernel and circular internal state, each stage-task has comparable computational and memory cost across stages, which helps balance the pipeline and reduces load imbalance.

OpenMP is a convenient framework for this design because it supports task parallelism with dataflow dependencies. Tasks are created inside a parallel region so that a team of worker threads is available to execute them. A single construct is used so that exactly one thread enqueues the tasks (avoiding duplicated task creation), while all threads participate in executing the generated tasks. Nevertheless, the proposed parallelization is not conceptually tied to OpenMP. The essential element is the stage–block task graph and its dependency structure: each block must traverse the cascade in order, and blocks belonging to the same stage must preserve the sequential update of the stage state. Other task runtimes or thread-pool mechanisms could implement the same dataflow using dependency counters, queues, or futures, although the scheduling policy and real-time guarantees would then depend on the particular execution environment. Correctness and efficiency depend critically on expressing two types of dependencies: the inter-stage dependency that enforces the cascade order for each block, and the intra-stage dependency that serializes access to the stage state to preserve the streaming semantics.

4.1. Pipeline Task Graph

Algorithm 2 summarizes the task graph induced by a task-parallel pipeline and makes explicit that the computation is expressed through three task types (visible in the three **spawn task** statements): LoadBlock to inject each input block into the pipeline (lines 11–12), StageBlockTask to apply one filter stage to one block and forward the low-pass block to the next stage (lines 15–16), and ReduceOutputSegment to merge thread-local partial outputs into the global output for the corresponding time segment (lines 18–19). These tasks operate on blocks of B samples and collectively implement the same blocked (blocks–stages–samples) structure as in the sequential schedule, while enabling multiple blocks to be processed concurrently at different stages of the cascade.

Fig. 2 illustrates the initial filling of the task-parallel pipeline. Each column corresponds to one stage of the multistage filter bank,

and each row represents a successive scheduling step. A block first enters the pipeline through a load task, and then advances through the cascade by means of stage–block tasks as soon as both its inter-stage dependency and the corresponding stage-state dependency are satisfied. During the first steps, only part of the pipeline can be active because earlier blocks have not yet reached the deeper stages. Once the first block has completed the last filtering stage, the corresponding reduction task can merge its thread-local contributions into the global output. After this filling phase, several blocks are simultaneously in flight at different stages, exposing the pipeline parallelism exploited by the OpenMP task scheduler.

The parallel efficiency of this pipeline heavily relies on the load balancing among its constituent tasks. Although the pipeline includes heterogeneous tasks such as data loading, filtering, and output reduction, the computational load is predominantly dictated by the N_f filtering stage tasks. The initial load task is extremely lightweight, involving only a contiguous memory transfer, whereas the N_f stage tasks exhibit nearly identical execution times; they all perform the same number of multiply-accumulate operations defined by the prototype filter length, except for a negligible computational difference during the result update of the final filter stage. It is worth noting that the final reduction task performs local accumulation across thread-private buffers, meaning its computational cost grows linearly with the number of threads. Nevertheless, as long as N_f is sufficiently large and dominates the per-block workload, the pipeline remains almost uniformly balanced, minimizing thread idle times during the steady-state regime.

Correctness and efficiency depend on a careful specification of data dependencies between tasks, expressed through `depend` clauses. The `depend(out:` $X_{0,b}$`)` on LOADBLOCK and the `depend(in:` $X_{f,b}$`, out:` $X_{f+1,b}$`)` on STAGEBLOCKTASK encode the inter-stage dataflow for each block, ensuring that stage $f+1$ for the same block starts only after stage f has produced the corresponding block output. In addition, the `depend(inout:` S_f`)` token serializes tasks belonging to the same stage across consecutive blocks, preserving the streaming semantics of the per-stage state. Finally, the reduction task is released only after the last stage has completed the block (`depend(in:` $X_{N_f,b}$`)` in line 19), which guarantees that all contributions relevant to that output segment are ready to be merged without requiring fine-grain synchronization in the hot path.

To minimize memory while keeping good locality, the intermediate block signals are stored in a small sliding window buffer of length W samples per stage, rather than in full-length arrays. The window is indexed circularly and reused across blocks. Choosing W as a power of two enables efficient modulo addressing (bitmasking), and selecting W large enough to hold the maximum number of simultaneously in-flight blocks prevents overwriting data that has not yet been consumed by downstream stages. The use of per-stage circular state inside LP_IFIR complements this design: the state footprint per stage is compact and repeatedly reused, improving cache residency.

4.2. Thread-local Outputs and Reduction

A direct parallel update of the global output buffer y would create race conditions because multiple tasks may attempt to update the same output indices concurrently. Using fine-grain synchronization (critical sections) or atomic operations inside UPDATEOUTPUT would serialize the hot path and severely limit scalability. Instead, each worker thread accumulates contributions into a private output vector $y_{\mathrm{local}}[t]$, and a separate reduction task periodically adds these private partial sums into the global output.

Algorithm 2 Parallel task-based pipeline filter

Require: Input $x[0..L-1]$, prototype taps $b[0..n_{\mathrm{coef}}-1]$, gains $g[0..N_f]$, block size B, threads T
Ensure: Output $y[0..L-1]$

1: $D \leftarrow (n_{\mathrm{coef}}-1)/2$; $y[\cdot] \leftarrow 0$; $n_{\mathrm{blocks}} \leftarrow \lceil L/B \rceil$
2: Choose W to hold in-flight blocks
3: Allocate sliding windows $Xwin[f][0..W-1]$ for $f = 0..N_f$ and initialize to 0
4: Initialize per-stage streaming states for $f = 0..N_f-1$
5: Allocate thread-local accumulators $y_{\mathrm{local}}[0..T-1][0..L-1] \leftarrow 0$
6: **parallel region** with T threads
7: **single** ▷ one thread enqueues tasks; all threads execute them
8: **for** $b = 0$ to $n_{\mathrm{blocks}}-1$ **do** ▷ Sample blocks
9: $i_{\mathrm{start}} \leftarrow bB$; $i_{\mathrm{end}} \leftarrow \min(i_{\mathrm{start}}+B, L)$
10: $B_c \leftarrow i_{\mathrm{end}} - i_{\mathrm{start}}$; $\omega \leftarrow i_{\mathrm{start}} \bmod W$
11: **spawn task** LOADBLOCK$(x, Xwin[0], i_{\mathrm{start}}, B_c, \omega)$
12: **depend(out:** $X_{0,b}$**)** ▷ writes into $Xwin[0][\omega..\omega+B_c-1]$ (circular in W)
13: **for** $f = 0$ to N_f-1 **do** ▷ Filter stages
14: $s \leftarrow 2^f$; $D_f \leftarrow D \cdot s$
15: **spawn task**
16: STAGEBLOCKTASK$(f, i_{\mathrm{start}}, B_c, \omega, W, b, g, L)$
17: **depend(in:** $X_{f,b}$**, out:** $X_{f+1,b}$**, inout:** S_f**)** ▷ S_f: token that serializes the stage state across blocks
18: **end for**
19: **spawn task**
20: REDUCEOUTPUTSEGMENT$(y, y_{\mathrm{local}}, i_{\mathrm{start}}, i_{\mathrm{end}})$
21: **depend(in:** $X_{N_f,b}$**)**
22: **end for**
23: **end single**
24: **end parallel region**

The filtering task body is shown in Algorithm 3. It reuses the same operators as the sequential description: it obtains the delayed stage-input sample via ReadDelay before applying LP_IFIR, and it invokes UPDATEOUTPUT unchanged, except that the destination is the thread-local buffer $y_{\mathrm{local}}[t]$. As in the sequential case, correctness relies on the ordering ReadDelay $\rightarrow$ LP_IFIR within each stage stream (Algorithm 3, lines 5–6), and the stage-state dependency in the pipeline schedule (Algorithm 2, line 12) ensures that blocks of the same stage execute in time order.

Reduction is performed in a streaming manner over the global output index. For each input block b, the reduction task flushes the segment $[i_{\mathrm{start}}, i_{\mathrm{end}})$ according to

$$y[p] \mathrel{+}= \sum_{t=0}^{T-1} y_{\mathrm{local}}[t][p], \quad y_{\mathrm{local}}[t][p] \leftarrow 0, \quad p \in [i_{\mathrm{start}}, i_{\mathrm{end}}).$$

Although UPDATEOUTPUT may place contributions at aligned indices $p = n + \Delta_f$, this flushing strategy remains correct because (i) all alignment shifts are nonnegative, and (ii) by the time block b completes the last stage, all tasks corresponding to input indices up to i_{end} (and all stages) have completed due to the transitive closure of pipeline and stage-state dependencies. Therefore, all contributions whose output indices lie in $[i_{\mathrm{start}}, i_{\mathrm{end}})$ are available in the thread-local buffers when the reduction task is triggered, while

Algorithm 3 STAGEBLOCKTASK: task body for one stage f and one block b

Require: Stage f, block start i_{start}, block length B_c, window offset ω, window size W, taps $b[\,]$, gains $g[\,]$, output length L
Ensure: Writes into $Xwin[f+1]$ and accumulates into $y_{\text{local}}[\cdot]$

$t \leftarrow \text{ThreadId}()$
for $j = 0$ to $B_c - 1$ **do** ▷ Samples in block
  $n \leftarrow i_{\text{start}} + j$
  $u \leftarrow (\omega + j) \bmod W$ ▷ circular in W
  $x_{\text{del}} \leftarrow \text{ReadDelay}(x^{(f)}, D_f)$
  $x_{\text{LP}} \leftarrow \text{LP\_IFIR}(Xwin[f][u], b, s)$
  $Xwin[f+1][u] \leftarrow x_{\text{LP}}$
  UPDATEOUTPUT($y_{\text{local}}[t]$, f, N_f, D, n, x_{del}, x_{LP}, g, L)
end for

contributions targeting future indices remain stored until their corresponding output segments are flushed by later reduction tasks.

5. EXPERIMENTAL ANALYSIS AND DISCUSSION

5.1. Experimental Environment

All experiments were conducted on an NVIDIA Jetson Orin Nano Developer Kit, which served as the target embedded platform throughout this work [3]. The developer kit integrates a Jetson Orin Nano 8 GB system-on-module (SoM) featuring a six-core Arm Cortex-A78AE CPU and an NVIDIA Ampere-class GPU with 1024 CUDA cores and 32 Tensor Cores. The SoM includes 8 GB of 128-bit LPDDR5 DRAM, with a theoretical peak bandwidth of 68 GB/s shared between CPU and GPU, and supports selectable power profiles (e.g., 7 W and 15 W).

The software environment was based on Ubuntu 20.04.5 LTS integrated within the NVIDIA JetPack 5.1.1 (L4T 35.3.1) software stack. The proposed algorithms were implemented in C and compiled using GCC 9.4.0 with high-level optimizations (`-O3` and `-march=native`). Parallel execution was implemented using OpenMP 4.5, specifically leveraging its task-based programming model to manage the data dependencies between the cascaded stages of the filter bank, enabling an asynchronous pipeline that maximizes multicore throughput.

The experimental campaign focused on assessing the impact of enforcing fixed CPU clock frequencies on both task completion time and power consumption. Frequency configuration was performed through the NVIDIA Jetson Linux Board Support Package (BSP) [4]. Under the default setup, CPU and GPU clocks are controlled by the `ondemand` governor; however, for our measurements, this behavior was overridden, and the CPU frequency was explicitly pinned to predetermined values.

To capture power behavior, we employed the enhanced `pmlib` framework [15], which samples and records power readings from the platform's on-board sensors at 10 Hz. Unless otherwise stated, CPU power refers to dynamic power, computed by subtracting the idle baseline measured at the same frequency from the power measured during filtering. Energy (E) was computed as the product of average dynamic power (P_{avg}) and execution time (t). When reporting per-block results, we utilize millijoules (mJ) derived from measurements in watts and milliseconds.

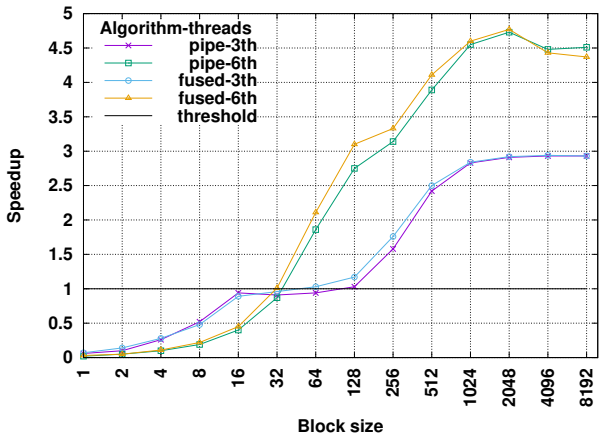

Figure 3: Speedup vs. block size for the task-pipeline and fused-load variants with 3 and 6 CPU threads. The threshold marks the sequential-baseline break-even point.

5.2. Experimental Results

5.2.1. Performance and Block-size Sensitivity

We first evaluate the sequential implementation. The results show that the total time required to filter one million samples is 0.81 s with a block size of 1, and increases only marginally to 0.84 s when the block size is raised to 16,384 samples. This corresponds to a throughput above 1.18 M samples/s for all the tested block sizes, confirming that the sequential baseline is already highly efficient.

The parallel version described in Section 4 can substantially improve throughput, but the attainable speedup depends strongly on the block size used to define each stage–block task. Fig. 3 reports the speedup of the original task-pipeline implementation and of a fused-load variant, in which the first filtering stage reads directly from the input buffer. The fused variant removes the separate LOADBLOCK task, and therefore eliminates one task, one memory copy, and one dependency per block, while preserving the same numerical output.

Figure 3 shows that very small blocks provide too little useful computation per task to compensate for the overheads associated with OpenMP task creation, dependency management, scheduling, and output reduction. For block sizes below 32 samples, the parallel implementations remain close to or below the sequential baseline. This effect is more pronounced with six threads than with three threads: the total number of stage–block tasks is determined by the number of blocks and stages, but using more worker threads can increase scheduling pressure, runtime contention, synchronization overhead, and the cost of reducing thread-local output buffers. Consequently, in this fine-grain regime, adding more threads does not necessarily improve performance.

Figure 3 reveals that, as block size increases, the task granularity becomes large enough for the pipeline to be effective. With 3 threads, both variants approach the ideal speedup of 3 for sufficiently large blocks, indicating that the stage-block decomposition exposes well-balanced pipeline parallelism when the overhead per task is amortized. With 6 threads, the speedup is lower than the ideal value of 6, but still exceeds 4.5 for the best block sizes. This confirms that the proposed pipeline exploits substantial parallelism despite the sequential state dependency within each filtering stage.

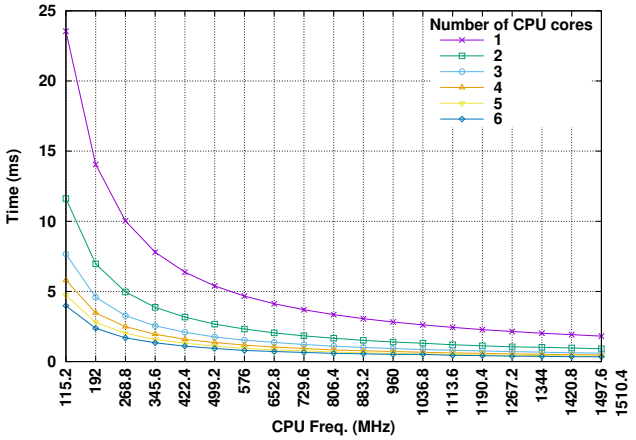

Figure 4: Processing time (ms) per 2048-sample block as a function of fixed CPU frequency, for different numbers of active CPU cores.

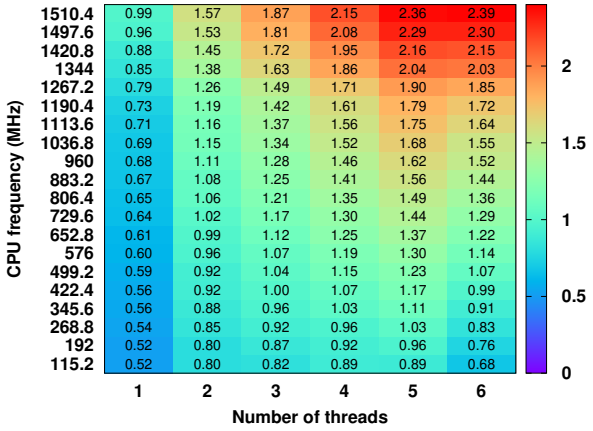

Figure 5: Average CPU power (W) during filtering as a function of CPU frequency and number of CPUs (idle baseline subtracted).

The fused-load variant mainly affects the small and intermediate block-size range. In this region, removing the explicit load task reduces part of the per-block overhead, so the fused variant tends to reach the useful-speedup regime earlier, especially with six threads. For larger blocks, however, the difference between the original and fused variants becomes small, because the filtering work performed inside each stage–block task dominates the cost of the additional load task. Thus, fusing the load operation is a valid implementation refinement, but it does not substantially alter the behavior of the pipeline in the high-throughput regime.

Figure 3 also illustrates the trade-off between task granularity and pipeline occupancy. Increasing the block size reduces the relative overhead of task management, but very large blocks also reduce the number of blocks available to keep the pipeline filled. In the tested configuration, block sizes of 1024 and 2048 samples provide the best balance between these two effects. We use a block size of 2048 in the remaining experiments because it provides high speedup with six threads while maintaining enough block-level concurrency for the pipeline.

5.2.2. Frequency Scaling, Power, and Energy

In edge deployments, maximizing throughput is not always the only goal; energy cost can be equally critical. A common approach to explore energy–performance trade-offs is to control the operating frequency of compute components [1, 5]. On Jetson Orin Nano, frequency can be left to a demand-based policy or pinned to fixed values. Here, we study the effect of fixing the CPU frequency to each of the 20 discrete values exposed by the platform, and quantify the impact on processing time and energy.

Fig. 4 shows that reducing CPU frequency increases the per-block filtering time for any number of cores. The slowdown is especially pronounced at the lowest three to four frequencies; beyond that, performance improves smoothly but with diminishing returns as the maximum frequency is approached. For instance, in the sequential case, the first 2048-sample block takes 23.5 ms at the minimum frequency (115.2 MHz) and 1.8 ms at the maximum frequency (1510.4 MHz), i.e., a 13× reduction in processing time. As more cores are enabled, the sensitivity to frequency decreases, and the minimum time is obtained with six cores at the maximum frequency, reaching 0.3 ms per block.

Average CPU power during filtering depends on both frequency and the number of active cores. Fig. 5 reports the (dynamic) CPU power for each combination. Power increases from 0.52 W (one core at the minimum frequency) to 2.39 W (six cores at the maximum frequency), with a gradual rise across the operating range. Overall, using one core gives the lowest CPU power at all tested frequencies, whereas increasing either the CPU frequency or the number of active cores raises the instantaneous power demand.

To assess the combined effect of time and power, we compute energy per block. The resulting energy map is shown in Fig. 6. Importantly, the configuration that minimizes energy is not the one that minimizes time nor the one that minimizes average power. The lowest energy observed is 0.43 mJ, obtained with the sequential version (one core) at 1036.8 MHz; neighbouring frequencies in the sequential configuration achieve similarly low values. Using all six cores at the same frequency range yields comparable energies (around 0.5 mJ), indicating that the reduction in execution time can compensate for the higher instantaneous power. The energy surface is also clearly non-monotonic: operating at very low frequency can increase energy because execution time grows faster than power decreases, a well-known effect in dynamic voltage/frequency scaling (DVFS) [1, 5]. This is particularly evident for two cores at the minimum frequency, where energy peaks at 3.60 mJ, but decreases rapidly either by enabling less or more cores, or by increasing the CPU frequency.

These results are platform-dependent and should not be interpreted as a universal optimum. The proposed dataflow is portable at the algorithmic level, since the block-based schedule, compact per-stage state, and task pipeline are not specific to the Jetson Orin Nano. However, the optimal block size, number of threads, CPU frequency, and low-level FIR kernel may need to be retuned on platforms with a different balance between memory bandwidth, cache behavior, floating-point throughput, and task scheduling overhead. On processors where floating-point execution rather than memory traffic becomes dominant, optimizations such as coefficient-symmetry exploitation, SIMD vectorization, or alternative state layouts may become more relevant.

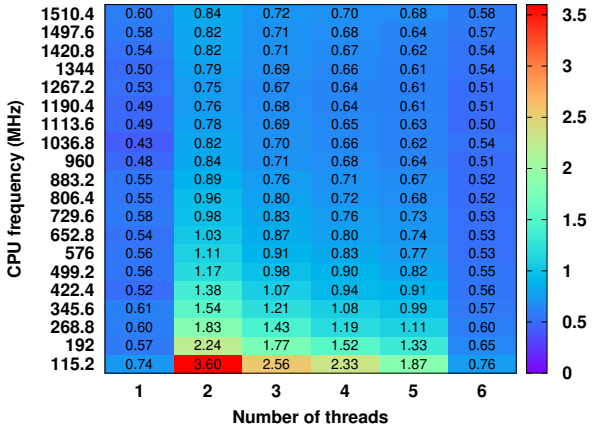

Figure 6: Total energy (mJ) consumed by the application in the CPU to filter each 2048-sample block as a function of fixed CPU frequency and number of CPU cores.

6. CONCLUSIONS

This paper studied the efficient realization of a multistage linear-phase octave filter bank on a CPU-based embedded edge platform. The results show that linear-phase equalization can be made practical when implementation choices account not only for arithmetic cost, but also for data movement, locality, state management, and parallel execution. The optimized sequential implementation already provides throughput well above standard audio sampling rates for single-channel processing, while the OpenMP task pipeline exploits the multistage structure for sufficiently large blocks, reaching speedups above $4.5\times$ on the Jetson Orin Nano.

The experimental evaluation shows that the most appropriate operating point is not the fastest one. Fixed-frequency measurements reveal a non-trivial trade-off between execution time, dynamic power, and energy per block. These results support the proposed high-throughput block-based kernel for edge-audio scenarios where phase preservation is important and the fixed delay can be tolerated, compensated, or traded for phase coherence. They also reinforce that efficient edge-audio processing should be tuned as a compromise between throughput and energy consumption.

Future work may extend this characterization beyond the block-based CPU kernel evaluated here. Relevant directions include callback-integrated scheduling with persistent pipeline state, governor-controlled execution with idle phases, SIMD-aware IFIR kernels with alternative state layouts, and mappings to dedicated audio DSPs or heterogeneous audio-processing platforms. These extensions require different implementation choices and experimental setups, but would complement the throughput–energy analysis presented in this work.

7. ACKNOWLEDGMENTS

This work was supported by the Spanish Ministry of Science and Innovation under projects PID2023-146569NB-C22 and PID2022-137048OA-C43, funded by MCIN/AEI/10.13039/501100011033 and "ERDF A way of making Europe", and by the Community of Madrid under R&D program TEC-2024/COM-360 (DISCO6G-CM).

8. REFERENCES

[1] C. Jiang, H. Zhang, Y. Ren, Z. Han, K.-C. Chen, and L. Hanzo, "Energy aware edge computing: A survey," *Computer Communications*, vol. 151, pp. 556–580, 2020.

[2] K. A. Swamy and Z. C. Alex, "Efficient low delay reconfigurable filter bank using parallel structure for hearing aid applications with IoT," *Personal and Ubiquitous Computing*, vol. 27, no. 3, pp. 1381–1394, 2023.

[3] L. S. Karumbunathan and S. H. Sheshadri, "Develop AI-Powered Robots, Smart Vision Systems, and More with NVIDIA Jetson Orin Nano Developer Kit," NVIDIA Tech. Blog. Available: https://developer.nvidia.com/blog/develop-ai-powered-robots-smart-vision-systems-and-more-with-nvidia-jetson-orin-nano-developer-kit/, Mar. 2023.

[4] NVIDIA, "NVIDIA Jetson Linux Developer Guide. Release 32.7.3," 2022, https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3273/.

[5] P. Haririan, "DVFS and its architectural simulation models for improving energy efficiency of complex embedded systems in early design phase," *Computers*, vol. 9, no. 1, p. Art. no. 2, 2020.

[6] P. Beckmann, J. Whitecar, J. Peil, and K. Hegde, "Achieving maximum audio processing throughput on the latest chipsets," in *Proc. AES Int. Conf. Automotive Audio*, 2022.

[7] V. Norilo, M. J. W. Verstraelen, and V. Välimäki, "Implementing a low-latency parallel graphic equalizer with heterogeneous computing," in *Proc. Int. Conf. Digital Audio Effects (DAFx)*, Trondheim, Norway, 2015, pp. 351–357.

[8] J. A. Belloch, J. M. Badia, G. León, B. Bank, and V. Välimäki, "Multicore implementation of a multichannel parallel graphic equalizer," *J. Supercomputing*, vol. 78, pp. 15 715–15 729, 2022.

[9] V. Bruschi, V. Välimäki, J. Liski, and S. Cecchi, "Linear-phase octave graphic equalizer," *J. Audio Eng. Soc.*, vol. 70, no. 6, pp. 435–445, Jun. 2022.

[10] U. Zölzer, Ed., *DAFx: Digital Audio Effects*, 2nd ed. Chichester, UK: John Wiley & Sons, 2011.

[11] 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.

[12] *OpenMP Application Programming Interface Version 4.5*, OpenMP Architecture Review Board, Nov. 2015. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[13] J. M. Badia *et al.*, "High-Performance Multistage Filter: C/OpenMP implementation," https://github.com/josembadia/hp-multistage-filter, 2026, accessed: 2026-06-05.

[14] V. Bruschi, V. Välimäki, J. Liski, and S. Cecchi, "A low-latency quasi-linear-phase octave graphic equalizer," in *Proc. Int. Conf. Digital Audio Effects (DAFx20in22)*, Vienna, Austria, Sep. 2022, pp. 94–100.

[15] S. Barrachina *et al.*, "An integrated framework for power-performance analysis of parallel scientific workloads," in *Proc. Third Int. Conf. Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY)*, 2013.