electronics

MDPI

Article

M3-AC: A Multi-Mode Multithread SoC FPGA Based Acoustic Camera

Jurgen Vandendriessche [1,*], Bruno da Silva [1,2,*], Lancelot Lhoest [1], An Braeken [1] and Abdellah Touhafi [1,2]

[1] Department of Engineering Sciences and Technology (INDI), Vrije Universiteit Brussel (VUB), 1050 Brussels, Belgium; Lancelot.charles.lhoest@vub.be (L.L.); an.braeken@vub.be (A.B.); abdellah.touhafi@vub.be (A.T.)
[2] Department of Electronics and Informatics (ETRO), Vrije Universiteit Brussel (VUB), 1050 Brussels, Belgium
* Correspondence: Jurgen.Vandendriessche@vub.be (J.V.); bruno.da.silva@vub.be (B.d.S.)

Abstract: Acoustic cameras allow the visualization of sound sources using microphone arrays and beamforming techniques. The required computational power increases with the number of microphones in the array, the acoustic images resolution, and in particular, when targeting real-time. Such a constraint limits the use of acoustic cameras in many wireless sensor network applications (surveillance, industrial monitoring, etc.). In this paper, we propose a multi-mode System-on-Chip (SoC) Field-Programmable Gate Arrays (FPGA) architecture capable to satisfy the high computational demand while providing wireless communication for remote control and monitoring. This architecture produces real-time acoustic images of 240 × 180 resolution scalable to 640 × 480 by exploiting the multithreading capabilities of the hard-core processor. Furthermore, timing cost for different operational modes and for different resolutions are investigated to maintain a real time system under Wireless Sensor Networks constraints.

Keywords: acoustic camera; SoC FPGA; Hardware–Software Co-Design, multi-mode; multithread; Delay-and-Sum beamforming; Wireless Sensor Networks

Citation: Vandendriessche, J.; da Silva, B.; Lhoest, L.; Braeken, A.; Touhafi, A. M3-AC: A Multi-Mode Multithread SoC FPGA Based Acoustic Camera. *Electronics* **2021**, *10*, 317. https://doi.org/10.3390/electronics10030317

Academic Editor: Alexander Barkalov
Received: 23 December 2020
Accepted: 22 January 2021
Published: 29 January 2021

1. Introduction

Acoustic cameras visualize the intensity of sound waves, which is used to be graphically represented as an acoustic heatmap, allowing the identification and localization of sound sources. Arrays of microphones are used to collect the acoustic information from certain beamed directions by applying beamforming techniques. Due to the high Input/Output (I/O) capability required to interface such microphone arrays, the high level of parallelism presented in such systems and the relatively low-power that Field-Programmable Gate Arrays (FPGAs) offer nowadays, most of the acoustic cameras use this technology to compute the needed operations for acoustic imaging. Combining these acoustic images with images from a Red-Green-Blue (RGB) camera adds another layer of information, which facilitates the identification and the localization of sound sources. The combination of both sensorial information demands, however, additional computational power to provide a near real-time response. Despite the fact that FPGAs are known for offering massive parallelism, power efficiency and low latency for streaming applications, they also demand a high design effort, which does not facilitate the addition of new operational modes.

Solutions based on standalone CPU do not provide the necessary parallelism that is required to generate real-time acoustic heatmaps out of the streamed signals coming from a microphone array. Furthermore, it also cannot provide the high number of I/O capabilities required to interface large arrays composed of tens of microphones like the SoundCompass [1]. On the other hand, although current FPGAs can satisfy the computational demands, the support of different operational modes would require the use of large

FPGAs due to the additional resource consumption. Moreover, further updates can still be limited by the resources of the FPGA.

Nowadays System-on-Chip (SoC) FPGA architectures are an alternative to single FPGAs. The combination of the high performance offered by FPGAs with the flexibility of a processor is here presented as a solution for real-time multi-mode acoustic cameras. Therefore, while the FPGA part processes the microphone signals to generate acoustic images, the processor not only performs additional image processing but also manages the wireless communication. In fact, the multi-mode support allows the adaptation for different scenarios like hand held devices or to deploy on nodes in Wireless Sensor Networks (WSNs). One of the most popular SoC FPGAs is the Xilinx Zynq, which does not only offer a relatively large amount of reconfigurable logic resources in the Programmable Logic (PL), but it also provides an ARM-based general purpose processor in the Processing System (PS) for fast software deployment. The communication between these two parts becomes a challenge to fully exploit the available bandwidth (BW) for real-time applications such as acoustic cameras. In order to maximize the throughput, an optimal distribution of the tasks between the FPGA and the CPU is required, while also minimizing the overhead in the communication.

This paper extends the work and results presented in [2]. On the one hand, the presented acoustic camera can combine visual and acoustic information by integrating an RGB camera into the system. As a result, new operational modes are supported. On the other hand, the multicore processor available in the target heterogeneous SoC is exploited by using multiple threads. It increases the flexibility of our acoustic camera to target different applications, like handheld devices or Wireless Sensor Networks (WSN), where low BW and power consumption are crucial. Therefore, the throughput of the supported modes is investigated and compared to the available bandwidth of Bluetooth Low Energy (BLE) in order to evaluate the feasibility of using the novel multi-mode multithread acoustic camera as a node in a WSN. These extensions result in a novel Multi-Mode Multithread Acoustic Camera (M3-AC). More specifically, we have improved the previous work with respect to the following aspects:

1. The proposed multithread approach exploits the multicore SoC FPGA in order to support new modes while providing real time response.
2. Combination of acoustics and visual information is now supported, enabling data fusion and additional image operations.
3. The original architecture has been optimized to almost double performance and to reduce the memory resource consumption on the FPGA part.

This paper is organized as follows. The state-of-the art of FPGA-based acoustic cameras is discussed in Section 2. An extensive comparison, regarding size of the microphone array, the acoustic image resolutions and the performance among others are here presented. The reconfigurable architecture of the M3-AC is described in Section 3. The balance of the image operations between the CPU and the FPGA is discussed. Firstly, the proposed architecture implemented on the FPGA to generate real-time acoustic images is described. Secondly, the operations performed on the CPU and the supported operational modes are presented. In Section 4, our multithread approach is discussed and compared to a single threaded approach. The multithread approach allows a different frame rate between the FPGA and the CPU, and improves the throughput by eliminating the need for a handshake mechanism between both parts. The evaluation of the M3-AC is done in Section 5. The optimized architecture is profiled in terms of performance, resource and power consumption. The timings for the different operations of each mode on the CPU are measured and compared to the frame rate and timing of the FPGA. Finally, conclusions are drawn in Section 6.

2. Related Work

Several FPGA-based acoustic cameras have been proposed in the last years. For instance, an acoustic camera able to reach acoustic image resolutions of 320 $\times$ 240 pixels is

proposed in [3], which embeds in a Xilinx Spartan 3E FPGA the operations to generate such acoustic heatmaps. Their architecture includes a Delay-and-Sum (DaS) beamfomer and a delay generator for the corresponding beamed directions, but it does not include any filter beyond the inner filtering during the Analog-to-Digital Converter (ADC) conversion of the incoming data from their analog Electrec Condenser Microphones (ECMs). As a result, their acoustic heatmap includes ultrasound acoustic information, reaching up to 42 kHz due to a missed high-pass filtering stage.

Digital Micro Electro-Mechanical Systems (MEMS) microphones are combined with FPGAs for robot-related acoustic imaging applications in [4]. Although their system supports up to 128 digital MEMS microphones due to the high I/O available in the chosen FPGA, only 44 microphones are used for their example. Their system reaches up to 60 FPS with an unspecified resolution, thanks to the multithread computation of the Fast Fourier Transform (FFT) required for the beamforming operations.

The authors in [5] propose an architecture using a customized filter together with DaS beamforming operations in the FPGA. Nevertheless, the authors do not provide further information about the power consumption, the timing or Frames Per Second (FPS) neither the output resolution, which is assumed to be 128 $\times$ 96 pixels as mentioned in [6].

The performance achieved with FPGA-based acoustic imaging systems has resulted in commercial products, as detailed in [7]. The authors describe a beamforming-based device (SM Instruments' Model SeeSV-S200 and SeeSV-S205) to detect squeak and rattle sources. The FPGA implements the beamforming stage, supporting up to 96 microphones and generating sound representations up to 25 FPS with an unspecified resolution.

The authors in [8] present a system to visually track auto vehicles and to characterize the acoustic environment in real-time. Their system is a microphone array composed of 80 MEMS microphones and a time-domain LCMV beamformer [9] embedded on an FPGA. The FPGA is only in charge of the filtering and the Filter-and-Sum beamforming operations. However, their resolution is only of 61 $\times$ 61 pixels reaching 31 FPS.

One of the few heterogeneous acoustic cameras is described in [10]. This system combines a SoC FPGA, a Graphics Processing Unit (GPU) and a computer desktop to generate acoustic images using a planar MEMS microphone array composed of 64 digital MEMS microphones. In the full embedded mode, the Xilinx Zynq 7010 performs the signal demodulation and filtering on the FPGA part while computing the beamforming operations on the embedded hard-processor. This acoustic imaging system is used to estimate the real position of the fan inside a fan matrix [11] and to create virtual microphone arrays for higher resolution acoustic images in [12]. Similarly, the authors in [13,14] implemented a 3D impulsive sound-source localization method on combining one FPGA with a Personal Computer (PC). The proposed system computes the DaS beamforming operation on the PC while the FPGA filters the acquired audio signals and displays through Video Graphics Array (VGA) the acoustic heatmap generated on the PC. However, there is no specifications about the resolution or achieved FPS.

An acoustic camera is proposed as part of a screen-based sport simulation in [15]. Their system combines an FPGA, which processes the audio signals and performs the DaS beamforming, a microcontroller, which detects the target type of sound and generates the steering vectors for the DaS beamformer, and a PC performing a specific sound recognition. Despite that the authors affirm the system operates in real time, they do not provide information about the FPS, number of steering vectors or resolution.

The authors in [16] propose a GPU-based acoustic camera combined with a special RGB camera. Although their architecture could achieve real time for lower resolutions, it only reaches around 2.8 FPS for the relatively high resolution of 672 $\times$ 376 pixels. Moreover, their solution is not scalable and it would present performance degradation when increasing the number of microphones.

Table 1 summarizes the most relevant features of the acoustic cameras. Parameters like the FPS or the acoustic image resolution, which determine the number of beamed directions, reflect the performance and the image quality of the FPGA-based architectures respectively.

Most of the architectures summarized in Table 1 do not only use an FPGA for the acoustic imaging operations, but combinations with other hardware accelerators like GPUs [10] or with multicore processors [13] to compute the beamforming operations, the filter's coefficients or to generate the visualization. Nevertheless, there is not a clear answer why recent FPGA-based acoustic cameras are not fully embedded like in [2] or in [3].

Table 1. Summary of architectures for acoustic imaging. The contributions described in this thesis are marked in bold.

Reference	Application	Year	Type of MIC	Model of MIC	MICs per Array	Device	Beamforming Algorithm	Resolution	Real-Time
[3]	Acoustic Imaging	2010	Analog ECM	Ekulit EMY-63M/P	32	Xilinx Spartan-3E XC3S500E	Time-Domain Delay-and-Sum	320 × 240	10 FPS
[4]	Robotic Applications	2012	Digital MEMS	Not Specified	44	Xilinx Spartan-6 LX45	Frequency-Domain Generalized Inverse	Not Specified	60 FPS
[5,6]	Acoustic Imaging	2014	Digital MEMS	Not Specified	32	Xilinx Spartan-6 XC6SLX16	Time-Domain Delay-and-Sum	128 × 96	Not Specified
[7]	Detection squeak and rattle sources	2014	Digital MEMS	Analog Devices ADMP 441	30 or 96	National Instruments sbRIO or FlexRIO (Xilinx Zynq 7020)	Time-Domain Unspecified Beamforming	Not Specified	25 FPS
[8]	Acoustic Imaging	2015	Analog MEMS	InvenSense ICS 40720	80	Xilinx Virtex-7 VC707	Linearly Constrained Minimum Variance	61 × 61	31 FPS
[10–12]	Acoustic Imaging	2016	Digital MEMS	ST Microlectronics MP34DT01	64	National Instruments myRIO (Xilinx Zynq 7010)	Frequency-Domain Wideband	40 × 40	33.4 ms to 257.3 ms
[13,14],	Acoustic Imaging	2017	Analog MEMS	ST Microlectronics MP33AB01	25	Xilinx Artix-7 XC7A100T	Time-Domain Delay-and-Sum	Not Specified	Not Specified
[2]	Acoustic Imaging	2018	Digital MEMS	Knowles Acoustics SPH0641LU4H	12	Xilinx Zynq 7020	Time-Domain Delay-and-Sum	160 × 120 up to 640 × 480	32.5 FPS to 1.8 FPS
[15]	Screen-based Sports Simulator	2019	Analog MEMS	Knowles SPH1642HT5H-1	13 up to 49	Xilinx Artix-7 A200T	Time-Domain Delay-and-Sum	Not Specified	Not Specified
[16]	Acoustic Imaging	2020	Digital MEMS	TDK InvenSense ICS-52000	10	GPU NVIDIA Jetson TX2	Time-Domain GCC-PHAT	672 × 376	2.8 FPS
Present Work	**Acoustic Imaging**	**2020**	**Digital MEMS**	**Knowles Acoustics SPH0641LU4H**	**12**	**Xilinx Zynq 7020**	**Time-Domain Delay-and-Sum**	**160 × 120 up to 640 × 480**	**61.8 FPS to 3.8 FPS**

Our M3-AC outperforms the discussed solutions not only in terms of resolution or performance (FPS) but also in flexibility thanks to supporting multiple modes. On the one hand, the M3-AC offers a variable range of resolutions, which can be changed on demand. The cost to pay is the reduction of the achievable FPS when increasing the resolutions. On the other hand, the multi-mode capability that the M3-AC offers does not only cover the output resolutions and the FPS but also the use of different operations to perform the identification of Regions-Of-Interest (ROIs) or the image compression to satisfy dynamic WSN context demands. Furthermore, few of the discussed acoustic camera solutions provide data fusion with other types of sensors such as our M3-AC does with the RGB camera.

3. A Multi-Mode SoC FPGA-Based Acoustic Camera System

The M3-AC system intends to exploit the combination of the Programming System (PS) and the Programmable Logic (PL) components of the SoC FPGAs to extend the use of acoustic cameras in WSN-related applications. SoC FPGAs such as Xilinx Zynq devices are composed by an FPGA part referred as (PL) and an ARM-based multicore processor referred as Programming System (PS). While the reconfigurable logic on the PL

part satisfies the low-power demands of WSNs, it also provides enough computational power to produce acoustic images in real-time. On the other hand, the PS part not only provides the necessary control to interface WSNs but also the flexibility to support multiple configurations without the need to partially reconfigure the FPGA logic. The computational balance between both components presents, however, several trade-offs that must be analyzed before reaching the true potential of SoC FPGAs for this particular application. Moreover, the presented solution supports multiple operational modes, which are decided by the WSN and managed by the PS, to better respond to WSN's demands. The use of multithreading to extend the capabilities of the M3-AC system is described in Section 4.

3.1. FPGA-CPU Distribution

Figure 1 depicts the proposed distribution of the computations between the CPU and the FPGA part. The main components of the M3-AC system are the microphone array, the RGB camera, the FPGA and the CPU parts of the Zynq architecture, and the wireless communication. The microphone array and the FPGA part compose the *front-end* while the RGB camera, the CPU part is the *back-end*.This separation becomes defined by the functional separation between the generation of the acoustic image (*front-end*) and the acoustic image processing (*back-end*).

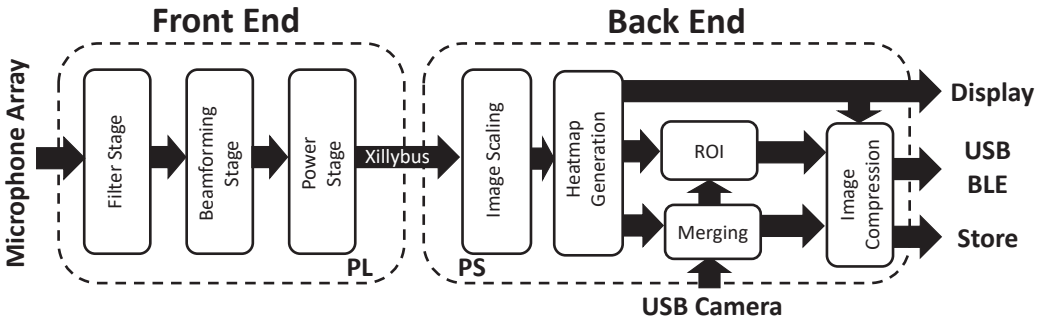

Figure 1. Distribution of the components into the Zynq 7020 device.

3.1.1. Proposed Front-End and Back-End

At the *front-end*, the FPGA part receives the acquired acoustic signal from the microphone array. The audio signal is retrieved from the microphones acquired signal after a filtering process performed in the filter stage. The beamforming stage aligns the audio signals in order to focus into a particular orientation determined by a steering vector while discriminating the inputs from other orientations [1]. In order to calculate the Steering Response Power (*SRP*), the output signal needs to be converted to an output power at the power stage. The *SRP* values obtained for each orientation are propagated to the CPU part to be represented as an acoustic heatmap. Xillybus simplifies the use of the AXI4 interfaces to transfer data from the FPGA part to the CPU [17], achieving experimental BW of 103MB/s [18]. On the FPGA side, a FIFO buffer is used to store the data, while on the CPU side, the data can be read by calling the read function like one would read from a file [19].

The *back-end* performs the local image processing, supports multiple image enhancements, interfaces the RGB camera and manages the WSN communication. The *SRP* values of the 3D beamforming are graphically represented in a heatmap format. The number of orientations or steering vectors (*No*) performed by the beamformer determines the heatmap resolution. While a low value of *No* leads to higher FPS, low resolutions are supported to satisfy the real-time constraints. The heatmap resolution is controlled by the WSN through the CPU, which adjusts the value of *No* on the FPGA to satisfy the WSN demands. This capability offers trade-offs in terms of performance and image resolution. Although a relatively low resolution acoustic heatmap is performed at the FPGA side to provide a real-time response, several image processing operations such as image scaling

are supported on the CPU part to improve the image resolution. Our multithreading approach enables multiple real-time image processing operations such as the generation of the heatmap from the values generated on the FPGA part, the scaling of the image, the RGB frame capturing and the merging with acoustic heatmaps, the identification of ROIs and the image compression. As a result, multiple modes are supported in order to adapt the image operations on the CPU part and to adjust the heatmap resolution on the FPGA side to satisfy the WSN demands. For instance, sound sources can be identified in the heatmap, where ROIs are marked based on predefined amplitude thresholds to be lately profiled. The identified ROIs and their coordinates are compressed and sent to the wireless network, reducing the overall BW consumption.

3.1.2. Distribution of the Roles

The distribution of the operations between the CPU and the FPGA is not a trivial task. The proposed computational balance between both technologies is motivated by several factors.

OpenCV Support

Despite the achievable performance on FPGAs encourages the implementation of most of the image processing operations on the FPGA, the support of all modes would result in a significant area demand. On the other hand, the processing of the acoustic heatmap on the CPU allows full usage of the advantages of the OpenCV library and functions. Although it is possible to use Xilinx HLS to implement certain image operations on the FPGA, not all openCV functions are fully supported yet. Table 2 provides a brief comparison of the support for the most relevant image operations used on the M3-AC system. For example, it is possible to upscale the image, but only three scaling modes are supported in Xilinx OpenCV (xfOpenCV) v2019.1 [20]. Similarly, relevant operations such as image compression techniques are not supported by the xfOpenCV library. The CPU provides enough flexibility to enable or disable different operations in the processing chain while supporting multiple operational and output modes. Moreover, the support provided by xfOpenCV limits the porting of required image processing operations to the FPGA, increasing the design effort for any future extension of the M3-AC system.

Table 2. Comparison of supported image operations in OpenCV 4.4.0 and xfOpenCV 2019.1 library.

Image Operations	Functions/Parameters	OpenCV 4.4.0 [21]	xfOpenCV 2019.1 [20]
Resize	Nearest neighbor	X	X
	Bilinear	X	X
	Bicubic	X	
	Area	X	X
	Lanczos	X	
Heatmap color	applyColorMap	X	X
Overlay	addWeighted	X	X
Edge detection	cvtColor	X	X
	Blur	X	X
	Canny edge	X	X
ROI	Threshold	X	X
	findCountours	X	
	boundingRect	X	
Image compression	PNG	X	
	JPG	X	

Computational Load

From the point of view of the computational demand, the proposed distribution between the FPGA and the CPU part intends to allocate on the reconfigurable logic the computational workload related to the acoustic heatmaps generation. Since the proposed architecture does not operate with floating-point data representation, the computational demand can be expressed in OPerations per Second (OPS), which is a much more suitable performance unit for FPGAs. Although the number of microphones in the array has a direct impact on the computational demands due to the audio recovering and filtering, the computational workload rounds 380 MOPS. The major computational demand comes, in fact, from the beamforming operation. Due to the generation of relatively large acoustic heatmaps in real time, the latest stages of the architecture operate at 100 MHz, requiring up to 1.4 GOPS. As a result, the standalone acoustic heatmaps generation demands around 1.78 GOPS, without considering the additional image processing operations such as edge detection or data fusion operations performed on the CPU part. Moreover, the FPGAs are not only well-known to satisfy real-time demands of signal processing applications, but also due to their power efficiency. For instance, while the CPU standalone consumes a minimum of 1.8 W when activated, our architecture demands a few hundreds of mW running on the FPGA. A simple comparison between the FPGA and the CPU of the Zynq for this application shows that the FPGA offers up to 6 times better performance per Watt than the CPU, with values up to 8.91 GOPS/W and 1.48 GOPS/W for the FPGA and the CPU respectively [22]. Such a power efficiency encourages us to embed on the FPGA the acoustic heatmaps generation while running on the CPU other image processing operations defined by the selected operational mode. Potential expansions of the M3-AC have also been considered. The architecture running on the reconfigurable logic is designed to easily scale when increasing the number of microphones, leading to a linear scaling in the computational demands. Such a scalability would not be possible by using a standalone CPU.

Bandwidth Demand

The BW demand at each stage is shown in Figure 2. The initial stages of the architecture present a BW demand of several tens of Mbps to retrieve and filter the audio signals of the 12 Pulse Density Modulation (PDM) MEMS microphones. The BW demands drastically increase after the beamforming stage due to operate at a higher frequency (further details are provided in Section 3.3). The lowest BW demand occurs after obtaining the *SRP* values. The upscaling of the acoustic heatmaps by factors of 2 or 4 significantly increases the overall BW demand. As a result, the most suitable distribution of the tasks between the FPGA and the CPU parts is the one depicted in Figure 1. While the *front-end* performs the minimum operations required to generate acoustic heatmaps on the FPGA, the flexibility of the CPU is used to support multiple operational modes on the *back-end*.

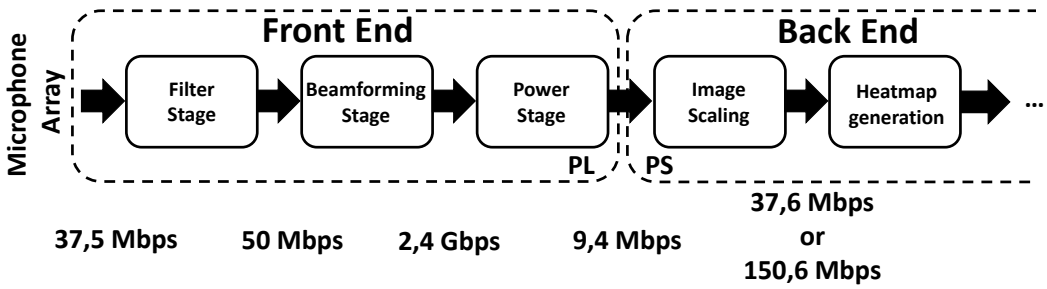

Figure 2. Bandwidth demanded by the first stages of the M3-AC system.

3.2. Front-End Description

3.2.1. Microphone Array and RGB Camera

The microphone array consists of 12 digital MEMS microphones SPH0641LU4H-1 [23] provided by Knowles placed in 2 sub-arrays (Figure 3). The inner sub-array is composed of 4 microphones placed at a radius of 20.32 mm from the center, whereas the outer 8 microphones are located at a distance of 40.64 mm from the center. The MEMS microphones SPH0641LU4H-1 [23] are selected due to their power efficiency when compared to the MEMS microphones ADMP521 [24] used in the microphone array described in our previous works [1,25]. The center of the Printed Circuit Board (PCB) has a hole with a diameter of 30 mm for mounting an RGB camera in the middle such that both the acoustic and video image can be overlapped.

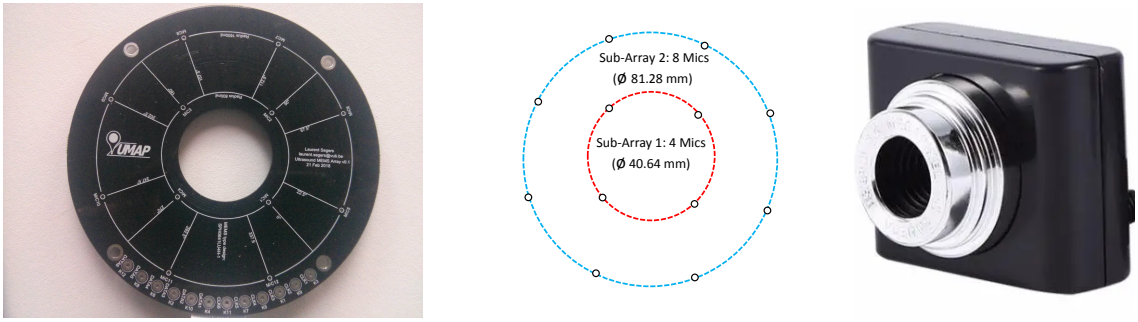

Figure 3. The microphone array (**left** and **center**) consists of 12 digital Micro Electro-Mechanical Systems (MEMS) microphones arranged in two concentric sub-arrays. The low-cost Universal Serial Bus (USB) Red-Green-Blue (RGB) Camera (**right**) achieves resolutions up to 640 × 480 pixels.

The output of the microphones is a PDM signal, which is internally obtained in each microphone by a ΣΔ modulator typically running between 1 and 3 MHz. Although analog microphones have been considered, digital MEMS microphones with PDM output present several advantages for microphone arrays [26]:

- The synchronization of the microphones is crucial in microphone arrays, forcing analog microphones to be synchronized at the ADC while digital MEMS PDM microphones simply use the same clock signal.
- The additional circuitry required for analog microphones reduces the level of integration when compared to digital MEMS microphones.
- The use of digital MEMS PDM microphones provides us an additional flexibility to explore alternative beamforming architectures. For instance, the architecture discussed in [27] provides a significantly better frequency response and a lower power consumption at the cost of a performance reduction.

The microphones are paired per sub-array such that 2 clocks and 6 data lines are required to interface the FPGA, which is done through one Peripheral Module (PMOD) connector. This sub-array approach enables the deactivation of all microphones of one sub-array by halting their clock signal. The shortest distance between the microphones is 23.20 mm and the longest distance equals 81.28 mm, which corresponds to acoustic frequencies ($\frac{\lambda}{2}$) of 7.392 kHz and 2.110 kHz respectively.

The RGB camera depicted in Figure 3 complements our microphone array. This low-cost Universal Serial Bus (USB) camera does not need a specific driver and is directly connected to the CPU part of the SoC FPGA. It achieves resolutions up to 640 × 480 pixels, which is used as upper limit to the resolution of the acoustic heatmaps.

3.2.2. Time-Domain Delay-and-Sum Beamforming Architecture

The reconfigurable architecture running on the FPGA part is based on the high-performance architecture presented in [25,28]. That DaS architecture offers a response fast enough to satisfy the performance demands of an acoustic camera. Unfortunately,

the price to pay is a small degradation in the accuracy of the beamforming, reflected in a relatively poor frequency response [29]. Instead, the proposed architecture achieves the same performance than the high-performance architecture while improving the frequency response. The architecture parameters are detailed in Table 3, is written in Hardware Description Language (VHDL) and implemented on the FPGA part using Vivado 2019.2.

Table 3. Configuration of the reconfigurable architecture under analysis.

Parameter	Definition	Value
F_s	Sampling Frequency	3.125 MHz
F_{min}	Minimum Frequency	1 kHz
F_{max}	Maximum Frequency	16.275 kHz
$BW_{Nyquist}$	Minimum BW to satisfy Nyquist	32.55 kHz
D_F	Decimation Factor	96
D_{CIC}	CIC Filter Decimation Factor	24
N_{CIC}	Order of the CIC Filter	4
D_{FIR}	FIR Filter Decimation Factor	4
N_{FIR}	Order of the FIR Filter	24

Figure 4 depicts the inner components of the three stages of the architecture implemented on the FPGA part. The complete architecture is processed in streaming mode and pipelines all the operations within each stage.

Filter Stage

The MEMS microphones of the array provide an oversampled PDM signal that needs to be processed to retrieve the original audio signal by demodulating the PDM signals. The required operations are performed in the filter stage, which is composed of multiple PDM demodulators or filter chains. Each microphone is associated to a filter chain, which is composed of a cascade of filters to reduce the signal BW and to remove the high frequency noise.

Nonetheless, this architecture, originally proposed in [28] and improved in [25], achieves a high performance at the cost of a higher resource consumption due to dedicate multiple filters to each microphone. The type and configuration of the filters are selected based on several design considerations related to the parameters such as the sampling frequency(F_s), the maximum supported frequency (F_{max}) and the decimation factor (D_F) summarized in Table 3. Further considerations about the design of the filter chain are largely discussed in [27], where a complete design-space exploration of possible PDM demodulators is presented.

The implemented filter chain is designed to operate in streaming and to minimize the resource consumption. For instance, the first filter is a 4th order (N_{CIC}) low pass Cascaded Integrator and Comb (CIC) decimator filter, which has a lower resource consumption since it only involves additions and subtractions. The CIC filter has a decimation factor (D_{CIC}) of 24 and it is followed by a moving average filter to remove the Direct Current (DC) offset introduced by the MEMS microphone. The last component of each filter chain is a 23rd order low-pass Finite Impulse Response (FIR) filter. The serial design of the FIR filter drastically reduces the resource consumption but forces the maximum order of the filter to be equal to the decimation factor of the CIC filter. The data representation used in the filter chain is a signed 32-bits fixed point representation with 16 bits as fractional part. Nevertheless, the bit width is increased inside the filters to minimize the quantization errors that the internal filter operations might have introduced. The data representation is set to signed 32-bits at the output of each filter by applying the proper adjustment and the FIR filter's coefficients are represented with 16 bits. The decimation factor of the FIR filter (D_{FIR}) is performed in the beamforming stage.

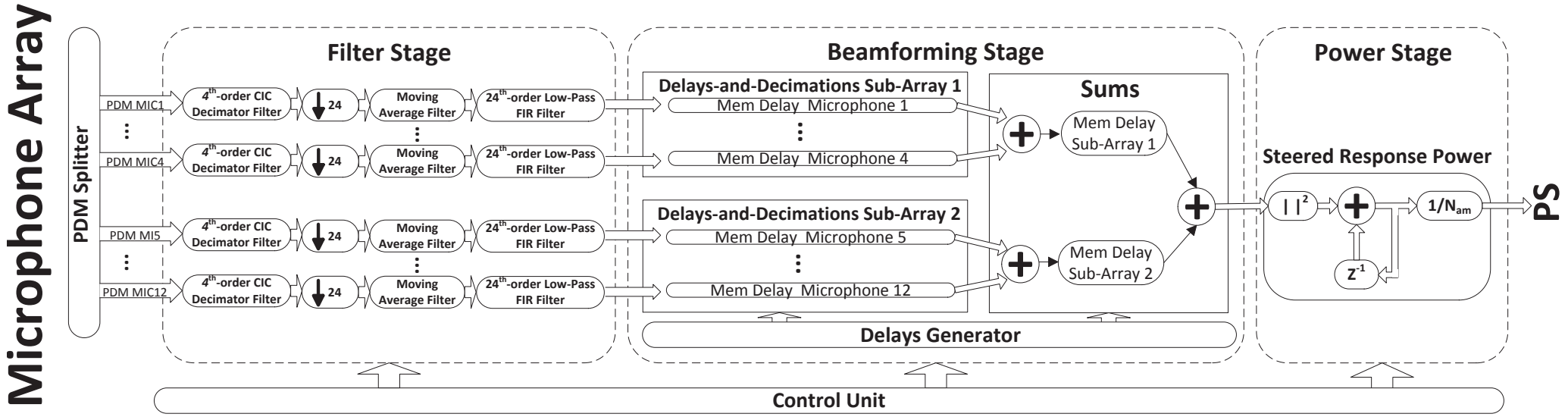

Figure 4. Overview of the FPGAs components. The PDM input signal is converted to audio in the cascade of filters. The Delay-Decimate-and-Sum beamformer is composed of several memories, associated to each sub-array to disable those memories linked to the deactivated microphones, to properly delay the input signal. The *SRP* is finally obtained per steering vector.

Beamforming Stage

The presented architecture uses the DaS beamforming technique to focus the array to thousands of steering vectors, which are determined based on the desired resolution of the acoustic image. The filtered audio from the filter stage is stored in banks of block memories acting as steering delays. The audio data is further delayed by a specific amount of time determined by the focus direction, the position vector of the microphone, and the speed of sound [1]. All possible delays are generated by the delays generator block and grouped based on the supported orientations. These delays are continuously generated in order to save memory resources (mostly BRAM), which otherwise, should be dedicated to store the precomputed delays generated during the compilation time [25,27].

In order to support a variable N_{am}, the implementation of the beamforming operation groups in sub-arrays the incoming signal of microphones. Therefore, the beamforming operation is only executed on the active sub-arrays, disabling all the operations associated to the inactive microphones in order to reduce the power consumption [27]. The overall memory required to perform DaS beforming technique rounds 73 kbits, with 65 kbits and 8 kbits to store the values from the outer and inner microphone sub-array respectively.

The DaS beamforming technique delays the input data a certain amount of time when beaming to a certain steering vector. These delays are calculated based on the steering vector, the sample frequency of the input data and the position of each microphone in the array [1]. Our acoustic camera uses an adapted hypercube distribution [30] to the Field-of-View (FoV) of the camera, which is 60°. A rectangular grid is taken in this section to calculate the euclidian distance between the positions of the microphones in the array and the points of the grid. These values are then normalized to obtain the vectors used to calculate the required delays. All calculations are done by the delays generator block in a continuous loop, which enables a multi-resolution support controlled by the CPU.

Power Stage

At the last stage, the delayed values from the beamforming stage are accumulated before the calculation of the SRP per orientation. The computation of SRP in the time domain for different steering vectors is used at the CPU part to generate the acoustic heatmaps. These steering vectors presenting a higher SRP correspond to the estimated location of the sound sources.

The heatmaps are displayed using a 24-bits RGB representation. Although, the heatmaps are generated on the CPU, a threshold of 255 is applied to the normalized SRP values to facilitate the generation of the heatmaps and the communication with the *back-end* by using the 8-bits Xillybus channel.

3.3. Trade-Offs

Architectures such as the one described in [25] support performance strategies to accelerate the generation of acoustic heatmaps. It has, however, a direct impact on the accuracy in terms of directivity (D_P) [31]. The range of the delays at the beamforming stage is inversely proportional to the sampling frequency at this stage. Like discussed in [27], the accuracy in architectures with high sampling rate [29] is higher than in architectures with lower sampling rate at the beamforming stage [25]. Nevertheless, the price to pay is the higher latency. Alternative architectures offer higher D_P [29] and a significantly lower power and resource consumption at the expense of performance. The proposed reconfigurable architecture is an intermediate solution where the highest performance is achieved while preserving a high level of accuracy.

3.3.1. Performance

The presented reconfigurable architecture solves the latency drawback by increasing the memory consumption at the beamforming stage. Compared to the architecture presented in [25], the proposed architecture locates the beamforming stage just between the low-pass FIR filter and the decimation operation, by combining the beamforming opera-

tion with a downsampling operation. During the beamforming operation, D_{FIR} values read from the BRAMs at the beamforming stage are discarded. The read operation of the beamforming memories has increments of D_{FIR}, which is equivalent to decimation. On the one hand, this solution increases by a factor of D_{FIR} the accuracy at the beamforming stage while performing like the architecture in [25] thanks to support of the same performance strategies. On the other hand, the memory requirements at the beamforming stage are increased by a factor of D_{FIR} due to all the undecimated filtered values that must be stored in the beamforming memories.

The performance strategy called continuous time multiplexing described in [25] is applied. Firstly, the PDM input signals are continuously filtered and converted to audio signals in the filter stage independently of the operations performed in the beamforming stage. Secondly, two clock regions are defined. While the filter stage and the storage of the audio samples in the beamforming stage are done at a F_S frequency, which also corresponds to the sampling frequency of the microphones, the calculation of the SRP values is done at a F_P frequency. Figure 5 depicts the two clock regions. The continuous time multiplexing strategy uses the beamforming stage memories to adapt the communication between the different clock regions. This strategy allow the beamforming stage to operate at a clock frequency F_P significantly higher than F_S. As a result, while F_S is 3.125 MHz (Table 3), F_P is assigned to 100 MHz. Notice that F_P doubles the frequency of the architecture presented in [2].

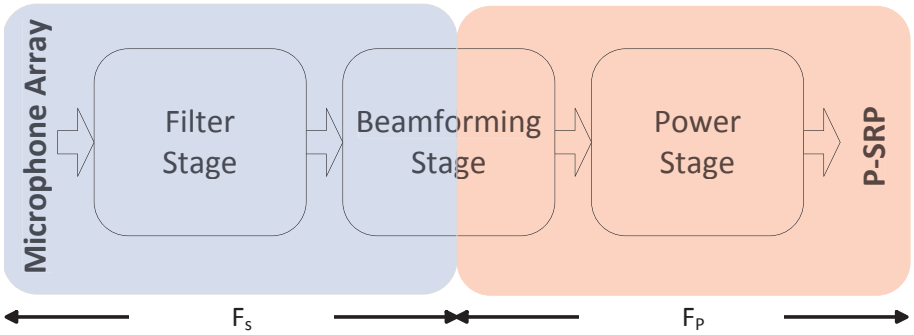

Figure 5. The proposed architecture operates at two different clock rates to achieve real-time acoustic imaging.

3.3.2. Frequency Response

A higher accuracy at the beamforming stage directly affects the overall frequency response of the architecture. Figure 6 depicts the comparison of the previous architectures and the proposed one. Each architecture has been evaluated for one sound source from 100 Hz to 12 kHz, with the same design parameters (F_s, D_F, etc.) as defined in Table 3 and considering 64 steering vectors in 2D for the SoundCompass microphone array [1]. The quality of the frequency response of each architecture is measured based on the directivity (D_P), which reflects the ratio between the main lobes surface and the total circle in a 2D polar map [31]. The average of all D_P along with the 95% confidence interval is calculated for 64 steering vectors. Moreover, the resulting D_P are based on the active sub-arrays of the original SoundCompass for the proposed architecture. The power-efficient architecture proposed in [29] is less sensitive per steering vector, presenting a lower variation on D_P as depicted in Figure 6 (top). This is the opposite for the high-performance architecture discussed in [25,28], whose value of D_P strongly depends on the steering vector. The reconfigurable architecture used here and depicted in Figure 6 (centre) offers a trade-off in terms of D_P, since its response is close to the accuracy obtained by the power-efficiency architecture proposed in [29]. There exists, however, slightly higher sensitivity per steered vector for sound source frequencies ranging from 8 to 10 kHz. As a result, the proposed reconfigurable architecture presents a slight degradation in D_P compared to the power-efficient architecture [29] while performing as fast as the

high-performance architecture [25,28]. Nevertheless, the cost is the additional memory consumption when compared to the architecture in [25,28] due to store D_{FIR} more delayed values per microphone.

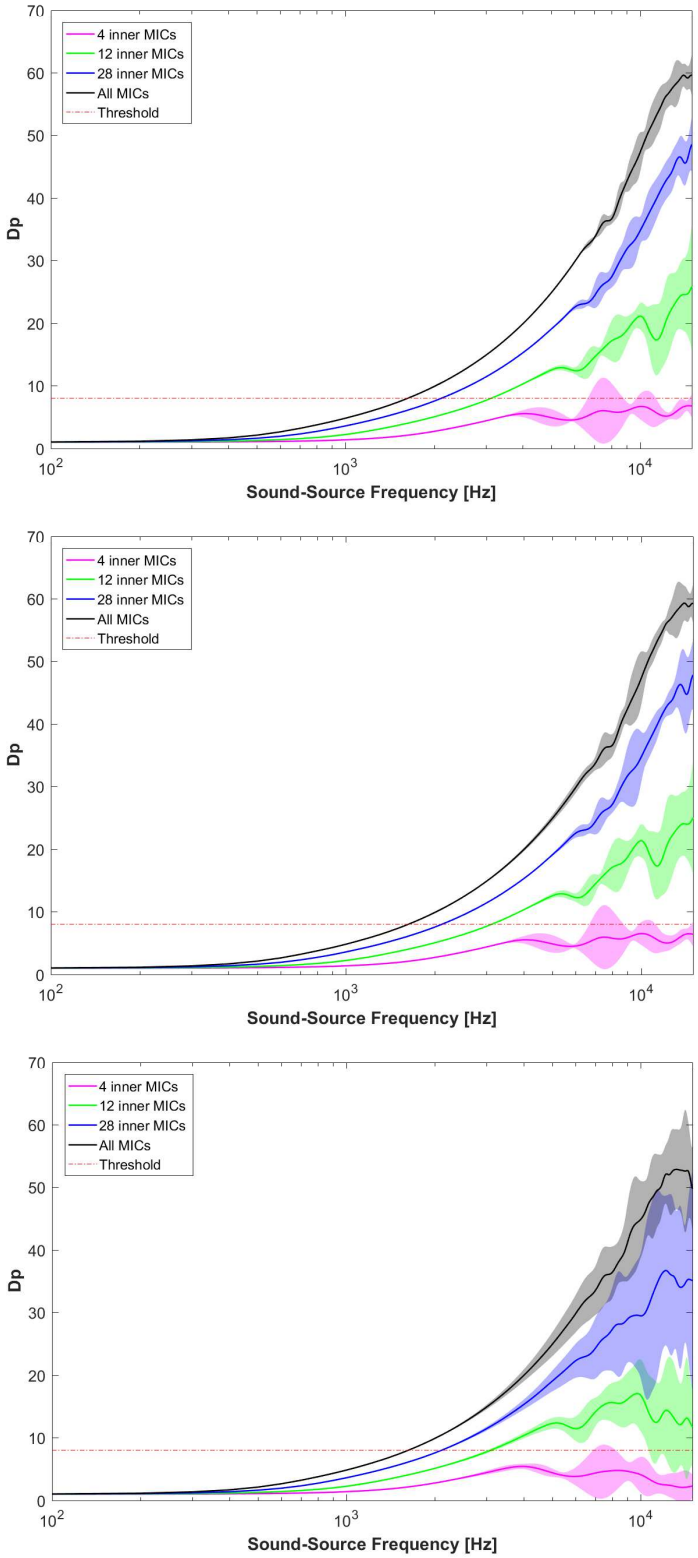

Figure 6. Comparison of the power-efficient architecture [29] (**top**), the presented architecture (**centre**) and the high-performance architecture [25,28] (**bottom**) using the 2D directivity [1] as metric done in [2]. The shadowed values represent the confidence interval for the 64 steered orientations.

Another property of a microphone array is the spatial resolution or Rayleigh criterion [32]. The spatial resolution is the minimum distance between two uncorrelated sound sources so that both sound sources can still be distinguished from each other. It depends on the size, shape of the microphone array and the frequency of the two sound sources. In Figure 7 three pairs of acoustic images containing two sound sources with different frequencies are depicted. Each image has a FOV of 60° in each direction and a resolution of 320x240. Both sound sources are placed at symmetric positions one meter from the center of the array. Notice how the spatial resolution increases if the frequency increases of either of the sound sources. The images are created using CABE, a Cloud-Based Acoustic Beamforming Emulator [33]. CABE allows the emulation of microphone arrays with fixed point and integer-based calculations in order to mimic the calculations that are performed on the FPGA, such as rounding errors. This emulator has been used as guidance of the development of the M3-AC architecture.

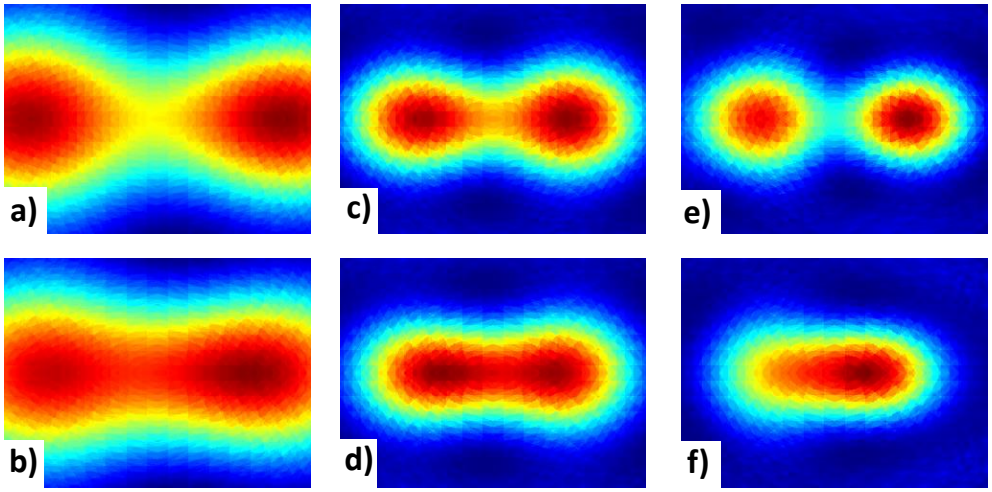

Figure 7. Acoustic heatmaps generated by an emulator to test the spatial resolution of the microphone array. Two sound sources are placed one meter from the array and moved towards each other to find the minimum distance between required between the sound sources to identify both sound sources. By increasing the frequency of either of the two sound sources, the minimum distance required to distinguish both sound sources from each other is decreases. Acoustic heatmaps: (**a**,**b**) two sound sources of 4.5 kHz and 5 kHz placed at (**a**) 62° and (**b**) 66°. (**c**,**d**) two sound sources of 8 kHz and 7.5 kHz placed at (**c**) 74° and (**d**) 78°. (**e**,**f**) two sound sources of 8 kHz and 9.5 kHz placed at (**e**) 74° and (**f**) 80°. 90° corresponds to the center of the image.

3.4. Back-End Description

The reconfigurable architecture is embedded on a Zynq 7020 SoC FPGA running Xillinux 2.0 [34], a Linux Operating System (OS) (Ubuntu 16.04) on the CPU part to enable a graphical use of the C++ OpenCV library (ver. 4.4) [35], which contains optimized functions for computer vision applications.

Figure 8 depicts our C++ OpenCV-based operational modes used by the CPU part to construct an acoustic heatmap from the FPGA data. The communication with the FPGA logic is via Xillybus [34], which is basically composed of First in First Out (FIFO) buffers. The CPU reads from the Xillybus FIFOs the SRP values generated on the FPGA. These values are placed in an $H \times W$ matrix, where H and W are the height and the width of the original heatmap resolution respectively. Once $H \times W$ SRP values are received, one acoustic heatmap frame is constructed. Depending on the selected operational mode, different OpenCV functions are used for the image enhancement, compression or data fusion with the data from the RGB camera.

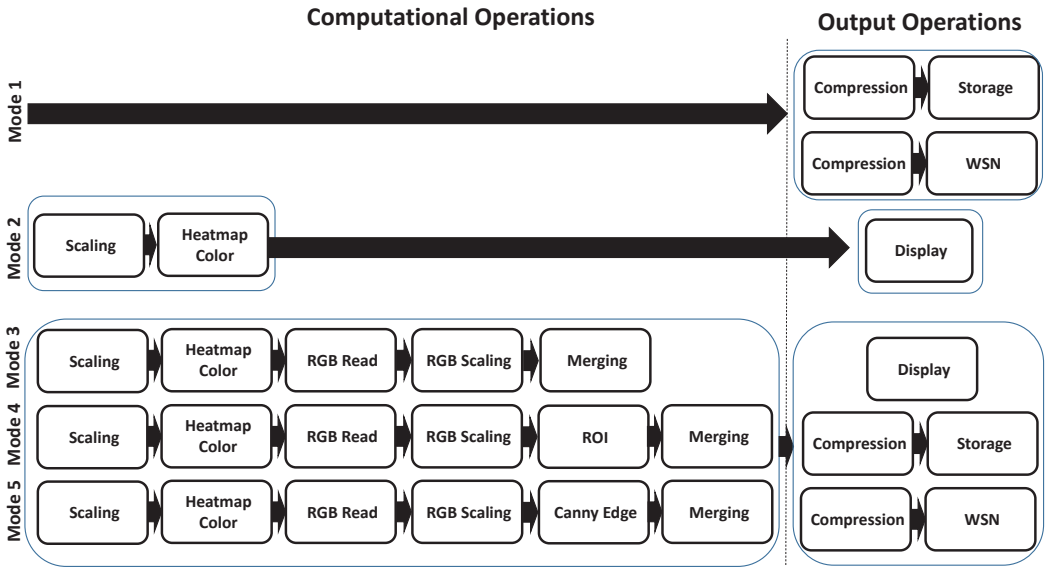

Figure 8. Overview of the image processing steps executed on the CPU. Multiple modes are supported to satisfy the most constrained Wireless Sensor Networks (WSN) demands.

3.4.1. Operational Modes

The architecture supports multiple operational modes based on the computational operations together with three different output modes as depicted in Figure 8. There exist five operational modes based on the computational operations, which involve operations related to the acoustic heatmap processing, the RGB image processing and the combination of both. The output modes vary for each supported mode, from enabling the local display, storage or wireless transmission for some of the supported modes. The first mode does no computations and only stores or transmits the received heatmap as raw values. Displaying the acoustic heatmap is not supported in this mode because the acoustic heatmap from the FPGA is expected to have a low resolution. The second mode enables the display of the acoustic heatmap by first scaling and applying a colormap. Notice that this mode does not support the wireless transmission or the local storage. Using the first mode to store/transmit the acoustic heatmap, and performing the scaling and coloring after loading/receiving the acoustic heatmap, will produce the same result, while the amount of data that is stored/transmitted is lower.

The other modes combine acoustic heatmaps and RGB frames. These operations add another layer of information. Displaying this combined frame does provide additional information, and for obvious reasons, less data will be send when both frames are sent overlapped instead of sending two separate images. The remaining three modes do support all possible output modes. Figure 9 shows the M3-AC system and outputs for some supported modes.

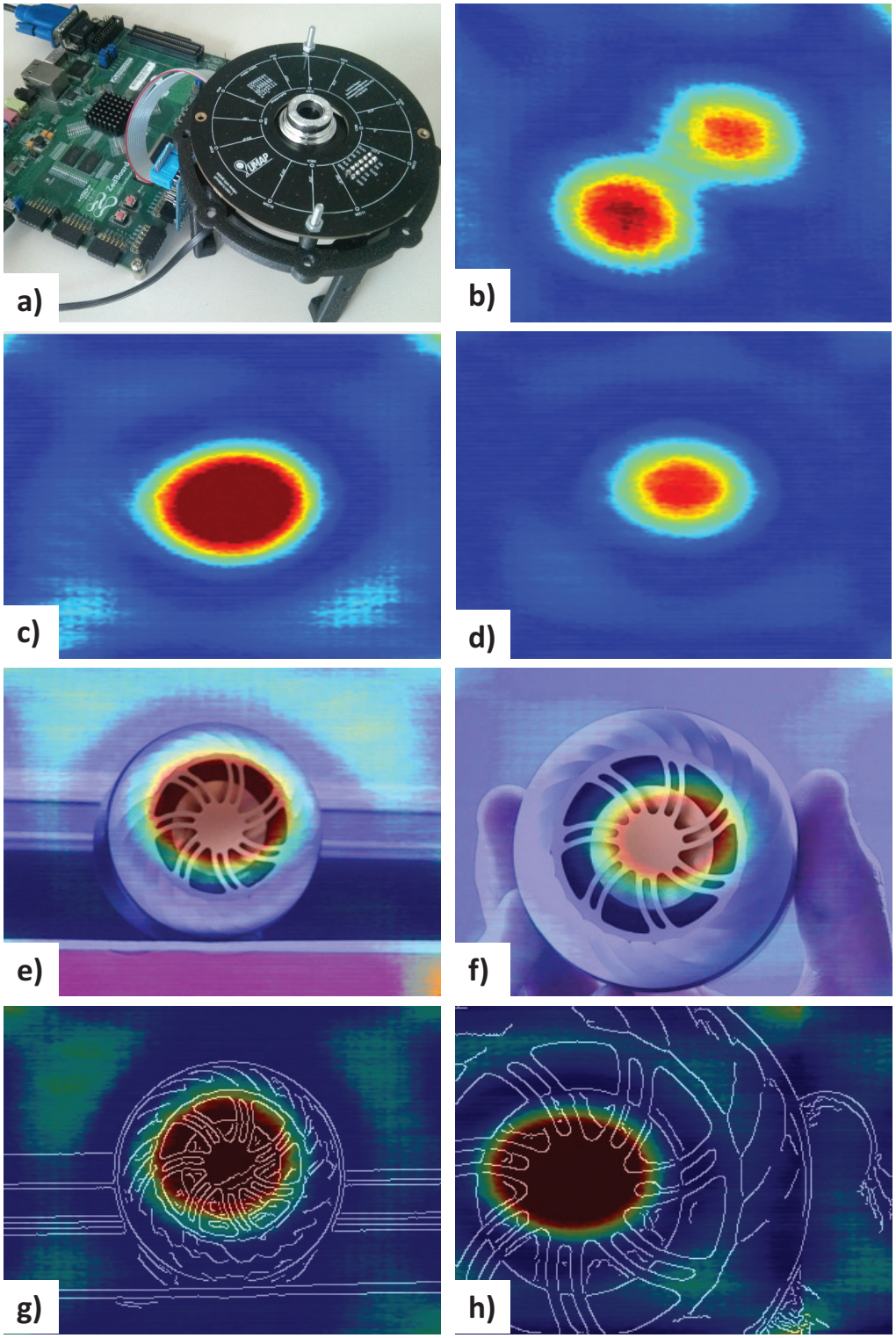

Figure 9. The M3-AC system consisting of the zedboard, microphone array and RGB camera used during the experiments (**a**). Acoustic hetamap of 320 × 240 with 2× scaling showing two sound sources (**b**), a single sound source of 4kHz half a meter (**c**) and one meter (**d**) away. The acoustic heatmaps can also be combined with unmodified images from the RGB camera (**e**,**f**) or after applying edge detection on the RGB frame (**g**,**h**).

3.4.2. WSN Communication

The Nordic nRF52840 USB dongle [36] is used for the WSN communication (Figure 10). This low-cost programmable USB dongle has been selected due to support Bluetooth 5.0, BLE, Thread, ZigBee, 802.15.4, ANT and 2.4 GHz proprietary protocols. Such a flexibility facilitates migration to a different wireless standard if the M3-AC is ported to a different WSN.

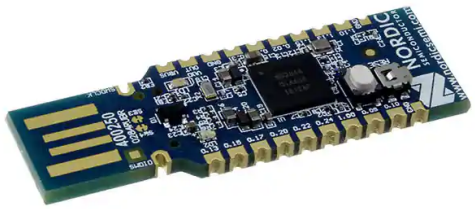

Figure 10. Nordic USB Dongle used for Bluetooth Low Energy (BLE) communication [36].

Due to the characteristics of the M3-AC, the default wireless standard is BLE. On the one hand, the M3-AC can generate images (e.g., acoustic heatmaps) with relatively large resolutions, and therefore, presents a high throughput. Whereas wireless standards based on IEEE 802.15.4 support maximum data rates around 250 kbps, BLE achieves data rates higher than 1 Mbps [37]. On the other hand, several modes of the M3-AC demand streaming transmissions, which is better supported in BLE. Nonetheless, further details regarding the throughput of the supported modes are discussed in Section 5.3.

For our M3-AC, the dongle is flashed with a softdevice, which contains the Nordic BLE code, and a Hexadecimal (HEX) file of the written code, generated with SEGGER Embedded Studio for ARM. Once programmed, the dongle is used as a Universal Asynchronous Receiver-Transmitter (UART) and an Application Programming Interface (API) written in C is used to interface the device. The CPU sends data to the dongle via serial communication, which in turn sends it to a connected device via BLE. String commands can be sent from the receiving device to the CPU, to change the operational mode of the CPU or to interrupt the communication. As a result, the benefits of using BLE are not only the available BW and the low power consumption [38], but also its presence on many devices such as laptops or smartphones. This allows a wide number of devices to operate as a base station, receiving the data coming from the M3-AC.

4. Multithread Approach

4.1. Single Thread Operational Mode Problem

In order to remain in real time and prevent a queue of unprocessed frames generated by the FPGA, the CPU has to process the frames faster than the FPGA generates these frames. The CPU performs the operations on a frame level basis. One frame is loaded and processed and then the next frame is loaded and processed. In a single threaded approach this means that the CPU will not read from the buffer while it is processing. The FPGA will still send acoustic pixels to the buffer, because it generates the acoustic pixels one at a time. As a result, the amount of data in the buffer over time changes according to Figure 11.

There are four different situations:

1. The data is not read from the buffer, the amount of data in the buffer grows at the same speed as the FPGA writes values to it.
2. The CPU is reading frames at a rate that is lower than the frame rate of the FPGA. There is an accumulation of unprocessed frames in the queue.
3. The CPU is reading frames at the same rate they are generated in the FPGA. At the exact moment the buffer contains one frame, the CPU reads that frame. The FPGA and CPU are synchronised.

4. The CPU is requesting frames at a higher rate than the frame rate of the FPGA. In this scenario, the CPU processes the frames faster than the FPGA can generate them. The CPU needs to wait for the FPGA to finish the generation of the next frame. Because the CPU cannot process a frame until it has received the full frame, it will send multiple requests for data to the buffer until it has received enough data.

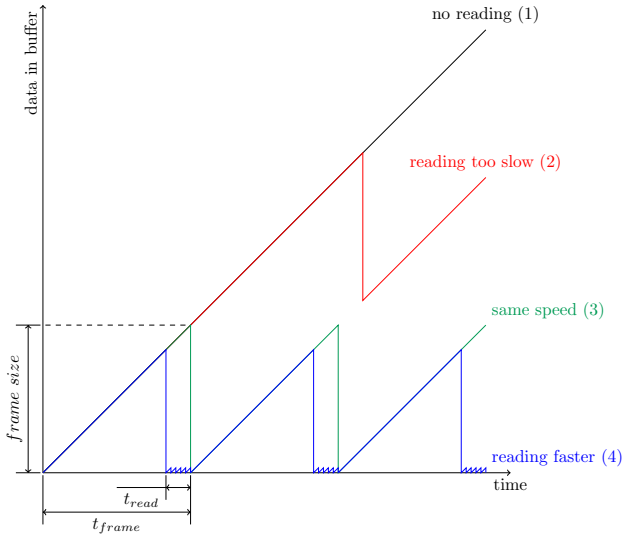

Figure 11. Data in the buffer for four different situations. (**1**): no data is read from the buffer and everything remains in the buffer. (**2**): data is read from the buffer, but at a slower rate than that it is stored. (**3**): Data is read from the buffer at the same speed it is stored. Every time one frame is ready in the buffer, it is read, resulting in an empty buffer. (**4**): A request for data is performed at a higher rate than that the data is stored in the buffer.

The first situation is unrealistic because no data is processed. The second situation will result in a system that is not real time because the amount of unprocessed acoustic pixels/frames will increase. Even more, because of the finite size of the buffer, the buffer will overflow over time. As a result, acoustic pixels are lost, hence losing acoustic information and faulty acoustic images. The third and fourth situations are the best: the CPU is processing the acoustic frames at the same speed or faster than the FPGA generates the frames. This relation can also be expressed as:

$$t_{process} <= t_{frame} - t_{minRead} \quad (1)$$

where $t_{process}$ is the time to process the frame by the CPU, t_{frame} is the time it takes the FPGA to generate one frame and $t_{minRead}$ is the minimum time needed to read one frame from the buffer by the CPU. If $t_{process}$ is below this threshold (situation 4), t_{read} will increase because the CPU cannot start processing the frame until it has received a full acoustic image. It limits the amount of supported modes when working at a lower resolution.

The finite size of the buffers further limits the processing time if the buffers cannot store a full frame. If the buffer can store m acoustic pixels and the frame consists of N_o acoustic pixels, with $m < N_o$, the buffer will overflow even in the third situation. Because there is no handshake mechanism between the FPGA and the CPU, the CPU will still read N_o acoustic pixels to form the acoustic image. However, this acoustic image will consist out of m acoustic pixels from one image and $N_o - m = n$ acoustic pixels from another acoustic image or images. These n acoustic pixels can be from another part of the acoustic image or the end of one acoustic image and the beginning of the next acoustic image. As a result, any acoustic image that is transferred from the FPGA to the CPU after a buffer overflow will also be misaligned. This means that the CPU must start reading from the buffer before the FPGA sends m acoustic pixels. For this reason, if the buffer cannot store a full frame,

$t_{process}$ is further limited by the time it takes to fill the buffer. This time corresponds to the size of the buffer divided by the the speed acoustic pixels are sent to the buffer (*dataRate*):

$$t_{overflowBuffer} = size(buffer) \times dataRate \tag{2}$$

this can be combined with Equation (1) to

$$t_{process} <= min(t_{frame} - t_{minRead}, t_{overflowBuffer}) \tag{3}$$

For low resolutions, $t_{process}$ is limited by t_{frame}, while for higher resolutions it is limited by the size of the buffer (see Figure 12). When using a buffer with a size of 16 kB, and a *dataRate* of one byte every 850 ns (see Section 5.1.2), it takes 13.6 ms to completely fill the empty buffer. This corresponds to the same time it takes the FPGA to generate an acoustic frame with a resolution of 146 × 109.

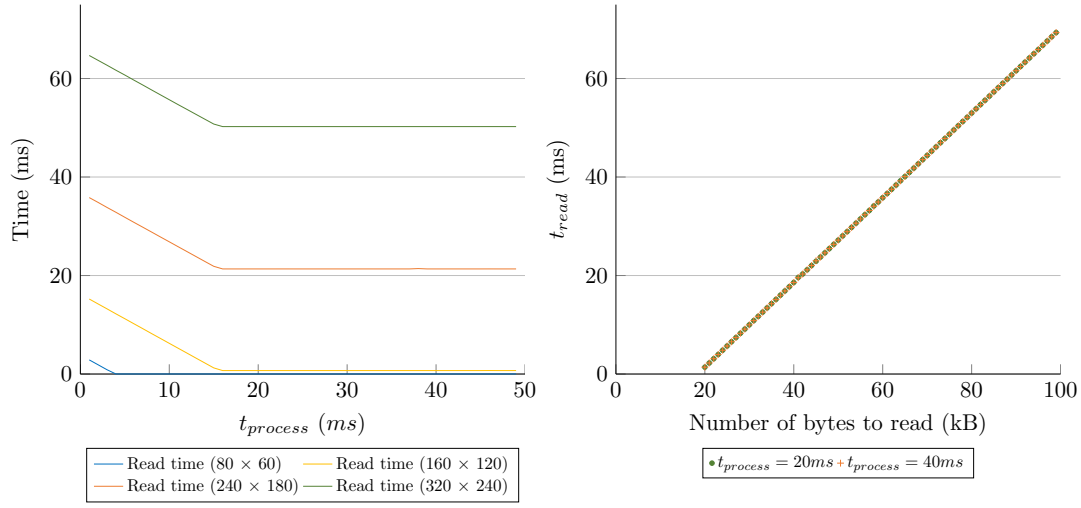

Figure 12. On the left, time to read data from the buffer for different resolutions. The function "usleep" is used to control $t_{process}$ for different resolutions. The read time for the higher resolutions decreases up to $t_{process}$ equal to 16 ms. After this, the read time becomes constant. This is due to the buffer containing the same amount of data when the read call starts (the maximum it can contain). On the right, time to read different amounts of bytes from the buffer. $t_{process}$ is kept constant to 20 ms and 40 ms and the read time is measured. The read times for both the 20 ms and 40 ms are the same for the same amount of bytes. The graphs has a linear trend line of $y = 0.8599x - 15806$ with $R^2 = 1$. One can see that this trend line gives a relationship between the size of the buffer (around 16 kB) and the time to generate one byte by the FPGA (0.85 µs).

4.2. Multithreading Approach as a Solution

To overcome this limitation on $t_{process}$, without consuming large amounts of resources to allocate a buffer that can store a full frame, a multithreaded approach is proposed. In this approach, which can be seen in Figure 13, a second thread is dedicated to reading the acoustic pixels from the buffer and storing them in application memory. Because the thread is continuously reading acoustic pixels from the buffer, the speed of set thread is determined by the speed of the FPGA. At the same time, the processing of the acoustic frames is performed on the first thread, which can have a different processing time ($t_{process}$). If $t_{process}$ exceeds t_{frame} the acoustic frames will no longer be misaligned even though there is no handshake mechanism between the FPGA and the CPU. Instead, some of the acoustic frames will be discarded and not be processed. This can be seen in Figure 14.

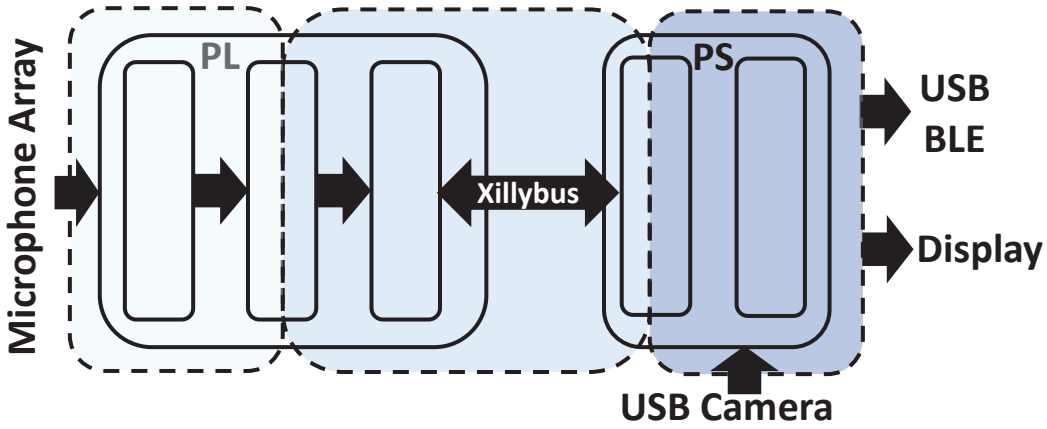

Figure 13. The proposed architecture operates at three different rates to reduce consumed resources and to optimize the execution time.

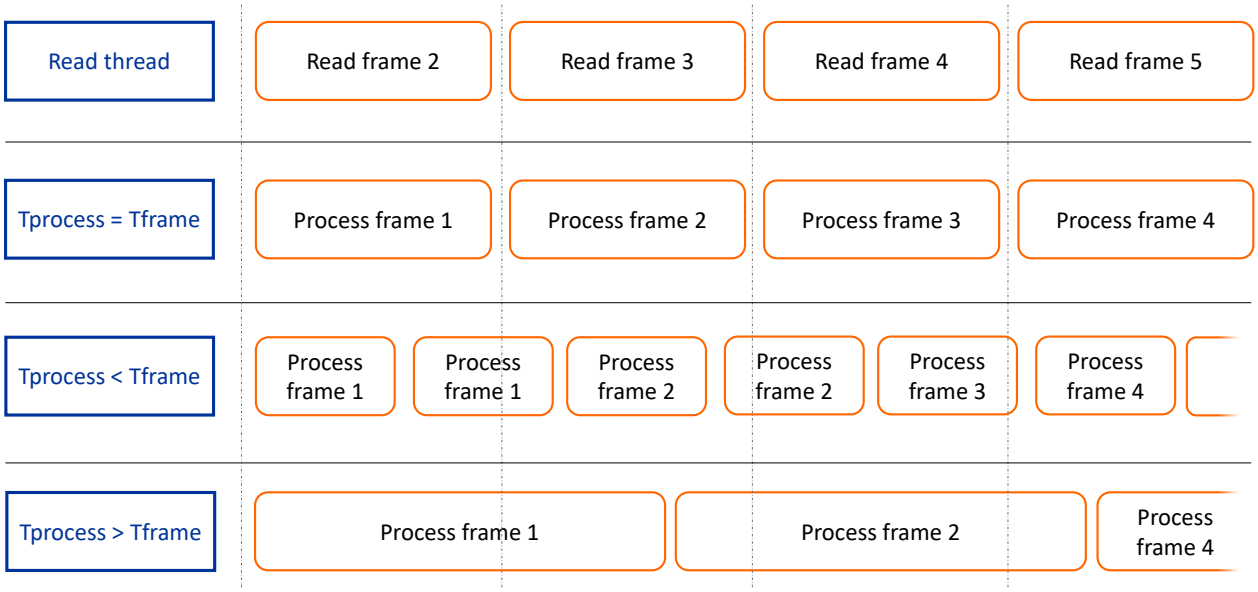

Figure 14. Processed frames for different situations, on top is the thread that reads the frames from the FPGA (stays synchronised with the FPGA), under that are three different situations where the frames are processed at the same speed ($FLR = 1$), faster ($FLR < 1$) or slower ($FLR > 1$) than the speed they are generated by the FPGA.

On top is the thread that reads the frames from the buffer. It is assumed that the first acoustic image is already read from the buffer and can be processed. Below that is the ideal situation where $t_{process}$ is equal to the time it takes the FPGA to generate the frame (t_{frame}) and the FPGA and the CPU have the same frame rate. Next is the situation where $t_{process}$ is less than t_{frame}. Here, certain frames are processed multiple times. The last situation is when $t_{process}$ is higher than t_{frame}. In this case, the CPU is not processing all frames but instead skips some frames generated by the FPGA. The Frame Loss Ratio (FLR) that describes the ratio between processed and generated frames can be expressed as

$$FLR = \frac{t_{process}}{t_{frame}} \tag{4}$$

In order to have an FLR close to 1, meaning that every acoustic frame is processed once and no acoustic frames are lost, one needs to have timings for all operations such that the resolution of the FPGA can be optimized depending on the selected mode. This can be found in Section 5.2.

In order to ensure that the read tread remains synchronised with the FPGA, the time between read calls is measured in the CPU. If the time between two read calls is too high, which causes a buffer overflow, the FPGA and read thread are reset. This prevents misalignment's of the acoustic frames. The maximum time a read call takes is determined by the resolution of the acoustic frames. As a redundant strategy, the read time is also compared to the minimum timings from Figure 12. If the read time is close to the minimum, it might be possible that the buffer was overflowing at the start of the read call and both CPU and the read thread are also reset.

5. Experimental Results

Our experiments evaluate the response of the microphone array, the resource consumption and the performance of the architecture and the overall performance of the system for WSN. Firstly, in Section 5.1 an analysis of the *front-end* regarding the timing, performance, resource and power consumption is presented and discussed. Secondly, in Section 5.2 a profiling of the OpenCV operations required for the supported modes is done. Finally, in Section 5.3 the supported modes are profiled and their support is discussed. Although these modes are evaluated in a stand-alone node without the WSN mote, we also provide experimental measurements of the USB BLE.

5.1. Analysis of the Front-End

5.1.1. Resource and Power Consumption

Table 4 summarizes the resource consumption on the FPGA reported by Vivado 2019.2 after the placement and routing. Although the filters have been designed to minimize the resource consumption, the filter stage has a dominant consumption of registers, LookUp Tables (LUTs) and Digital Signal Processors (DSPs) due to the streaming and pipelined implementation of the architecture. Despite the relatively large resource consumption of the presented reconfigurable architecture, it represents a lower resource demand when compared to the architecture in [2]. This reduction is due to the delays generator block, which generates at runtime the delays necessary to support thousands of steering vectors during the beamforming operation, reducing to half the demand of LUTs but increasing the DSP consumption. Thanks to this reduction on the resource consumption, microphone arrays composed up to 52 microphones (such as the one proposed in [1]) can be processed in parallel on a single Zynq 7020 SoC FPGA. The reduction of the LUTs consumption also enables the migration of the reconfigurable architecture to a more power efficient SoC FPGA devices, like the Flash-based SoC FPGA considered in [29]. Unfortunately, despite low-power Flash-based SoC FPGAs like the Microsemi's SmartFusion2 promise a low power consumption as low as few tens of mW, such devices embed an ARM Cortex-M3 microcontroller, which is not powerful enough to support the use of C++ OpenCV library.

Table 4. Zynq 7020 resource consumption after placement and routing of the proposed architecture when supporting any resolution between 20 × 15 and 640 × 480.

Resources	Available	Utilization				Percentage
		Filtering	Beam-forming	Delays Generator	Total	
Registers	106,400	10,600	1935	1578	14,331	13.35%
LUTs	53,200	8515	707	1682	10,999	20.67%
LUTs-FFs	53,200	7207	671	1593	9566	17.98%
BRAM18k	280	6	6	4	16	5.71%
DSP48	220	24	4	17	45	20.45%

The estimated power consumption reaches 1.95 W using Vivado 2019.2 power estimator tool, with around 1.8 W and 155 mW of dynamic and static power respectively. The power consumption is dominated by the CPU part, since the activation of the CPU leads to more than 1.68 W, whereas the FPGA part presents a power consumption of a

few hundreds of mW. Notice that the dynamic power consumption represents up to 92 % of the overall power consumption mainly due to operating at 100MHz to generate the SRP values.

5.1.2. Timing and Performance Analysis

The filtering and beamforming operations at the FPGA logic can be adjusted to generate acoustic heatmaps with different resolutions. The latency to process a single steering vector is determined by design parameters like the sensing time (t_s), F_s and D_F. The value of t_s, the time the microphone array is monitoring a particular steering vector, determines the probability of detection of sound sources under low Signal-to-Noise Ratio (SNR) conditions. Therefore, higher values of t_s improve the profiling of the acoustic environment by increasing the overall execution time per frame (t_{frame}). The proposed architecture calculates the SRP with $N_s = 64$ samples, which represents 6144 input PDM samples per steering vector for a $D_F = 96$ (Table 3). For the F_s described in the same table, $t_s \approx 1.96$ ms. The latency to calculate the SRP for one steering vector using $N_s = 64$ is 85 clock cycles, independently of the operational frequency at the power stage. This is possible thanks to storing in the steering delays of the delay-decimate-and-sum beamforming all the required samples to compute the SRP for one orientation.

The beamforming operation is performed at a higher clock frequency than F_s as the performance strategy called continuous time multiplexing described in [27]. The operational frequency F_P at the beamforming and power stage has been increased to 100 MHz, which corresponds to the Xillybus' clock frequency [34]. Therefore, the time to calculate the SRP per orientation (t_o) is approximately 0.85 µs.

Table 5 details some of the possible heatmap resolutions and the expected performance in FPS when operating at $F_P = 100$ MHz. In order to reach real time, the time per frame t_{frame} must be lower than 33.3 ms or 50 ms to reach 30 FPS or 25 FPS respectively. This requirement drastically reduces the maximum heatmap resolution to 240 × 180. On the other hand, in order to guarantee the independency of each acoustic heatmap, each acoustic image must be generated from the acquired acoustic information in a period higher than $t_s/2$. Therefore, at least 32 out of the 64 samples used to calculate SRP have not been already used to generate one acoustic image. The value of No, which represents the acoustic heatmap resolution, must be high enough to satisfy the independency condition. Therefore, it follows that $t_{frame} \geq t_s/2$. This condition limits the minimum supported resolution because it is only satisfied when $No > 1224$ based on the design parameter in Table 3 and by operating at 100 MHz. As a result, resolutions as low as 40 × 30 do not satisfy the independency condition.

Table 5. Timing (in ms) and Performance (in FPS) of the FPGA part based on the supported resolutions.

Resolution	No	t_{frame} [ms]	FPS
20 × 15	300	0.255	3921.5
40 × 30	1200	1.02	980.4
80 × 60	4800	4.08	246.1
160 × 120	19,200	16.32	61.8
240 × 180	38,400	32.65	30.6
320 × 240	76,800	65.28	15.3
640 × 480	307,200	261.12	3.8

5.2. Analysis of the Back-End: Individual Computational Operations

Like discussed in Section 4, a dedicated thread is used for receiving and storing the SRP values from the FPGA, which are used to generate the acoustic pixels in a heatmap format on the CPU. (Figure 14). A frame is defined as all acoustic pixels generated on the FPGA for a given resolution (Table 5). The data coming through the Xillybus is read, and stored into one of two memory arrays that are shared with the processing thread. The two memory arrays are used as double buffering. Processing of the acoustic heatmap

is done on the processing thread. In order to process all frames from the FPGA, it is important that the processing of one frame takes less time than it takes the PL to generate one frame ($FLR <= 1$). Computing intensive modes can be restricted to operate with lower resolutions to fulfill this requirement. For this reason, it is important to profile each operation in the processing chain for each mode. Some modes require more operations than others, e.g., a mode that does not use the RGB camera does not require to read, scale and combine a frame of the RGB camera with the heatmap from the FPGA. An overview of the operations used in each mode is here described. All timings are the result of 1000 measurements.

During the measurements, no other program was running and the CPU was in an idle state (CPU usage < 3%) before starting the test and after ending the test. The code is compiled with the GCC compiler using C++ 11 [39]. The -O3 option is used to turn on code optimizations such as loop unrolling, function inlining,.. and *pthread*, which leads to a higher speed.

5.2.1. Heatmap Scaling

Most of the modes combine a frame from the acoustic camera with a frame from an RGB camera. In that case, both frames must have the same resolution, and have the same data format: either black and white or color. For instance, the raw acoustic heatmap is only a black and white image, so either the frame from the RGB camera needs to be converted to black and white or the acoustic heatmap must be converted to color. Our platform used the latter of the two, scaling the acoustic heatmap to the desired resolution before coloring. This is done with the OpenCV function *resize*, which supports multiple modes for resizing the frame [40–42]:

1. *Nearest-neighbour*: This method has been discarded since it only selects the value of the nearest pixel without performing interpolation. Although being the fastest method, its output images are highly pixelated.
2. *Bilinear*: This method calculates a new pixel value by taking a weighted average of the four nearest neighbouring original pixel values. A smoother result than the *Nearest-neighbour* is obtained at the cost of undesired lines. Nonetheless, this method is the fastest of the other three.
3. *Bicubic*: This type of interpolation provides the best visual result, but also is the more time demanding algorithm. Each new pixel is calculated by the *Bicubic* function using the 16 pixels in the nearest 4×4 neighbourhood. The result is a smooth heatmap image.
4. *Lanczos*: This interpolation method is based on the sinc function but it demands roughly double the amount of time to resize an image than the *Bicubic* method. Although the result is closer to the *Bicubic* method, some artefacts might appear in the rescaled image.

For the sake of simplicity, our acoustic heatmaps are only scaled by a factor of two or four, for which timings can be found in Figure 15. It can be seen that *Bilinear* is the fastest for all resolutions, while *Lanczos* is twice as slow as *Bicubic* and even three times slower than *Bilinear*. This can be explained by the complexity of the different interpolation techniques as described before. The time to resize the frame is determined by the output resolution. As a consequence, when one of the modes is too slow the resolution of the FPGA can be increased to optimize FLR. This higher resolution means that the FPGA needs to generate more acoustic pixels, increasing the time to generate one frame. The CPU, on the other hand, will still require the same amount of time to process the frame. Also the opposite can be done. If the CPU is processing frames faster than the FPGA generates them, one could decrease the resolution of the FPGA.

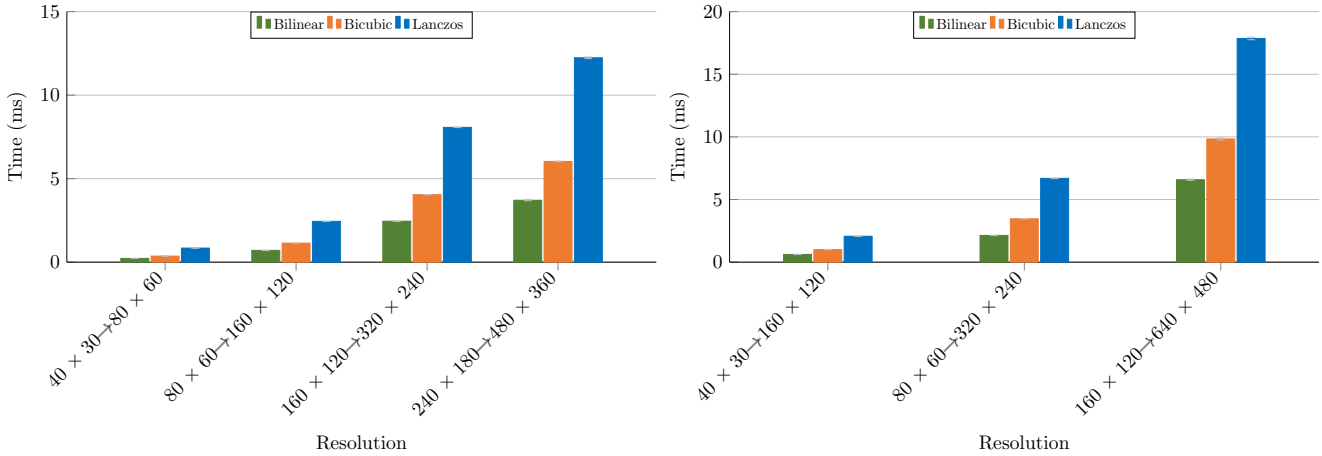

Figure 15. Time for rescaling the heatmap for different resolutions by a factor of 2 (**left**) and by a factor of 4 (**right**). The gray lines indicate the confidence interval.

5.2.2. Heatmap Color

In order to combine the acoustic heatmap with a frame from the RGB camera, it is desirable that both have the same color encoding. This means that the grayscale heatmap needs to be converted to RGB. This could be done by converting the colorspace. Another option is applying a colormap using the OpenCV function *applyColorMap* [43], which assigns each pixel a color based on the value of set pixel in the grayscale heatmap. The timing for this operation can be found in Table 6. Because each pixel in the output image of this function gets a color based on the value of the same pixel in the original image, the time increases linear with the number of pixels. However, for higher resolutions like 640 × 480, OpenCV uses multithreading to improve the speed [44]. It explains why the resolution 640 × 480 is not 4 times slower than 320 × 240. In some cases, it can be interesting to slightly increase the resolution so that the OpenCV uses multiple threads. Unfortunately, this is not our case since the resolution 640 × 480 is slower than 480 × 360. Moreover, it might be that a resolution of 520 × 400 is slower than 640 × 480. This also means that the code is executed on both cores and special care needs to be taken so that the read thread that is dedicated to reading out the FIFO is not stalled too long. If this is not the case, both cores can be used because the read thread spends most of its time in an idle state, waiting for values from the FPGA.

Table 6. Timing (in ms) for different operations on the CPU based on the supported resolutions together with the confidence intervals.

Resolution	#Pixels	Colormap [ms]	Canny Edge [ms]	Merging [ms]
80 × 60	4,800	3.775 ± 0.017	1.262 ± 0.017	0.772 ± 0.003
160 × 120	19,200	4.828 ± 0.027	3.463 ± 0.035	2.883 ± 0.003
320 × 240	76,800	9.510 ± 0.026	11.717 ± 0.076	12.422 ± 0.038
480 × 360	172,800	16.569 ± 0.024	23.274 ± 0.108	25.905 ± 0.067
640 × 480	307,200	17.801 ± 0.114	41.942 ± 0.146	45.961 ± 0.098

5.2.3. ROI

When working with higher resolutions, some modes can become too slow resulting in missed frames from the FPGA. To improve the overall timing it could be interesting to do a part of the processing on the full frame and then detect some ROIs. This results in a couple of time improvements. First of all, frames without any ROI can be discarded. If there is nothing interesting in the frame, the frame should not be processed, allowing to process the next frame faster and reducing the amount of processed data. A second reason why this improves timing is because it introduces the possibility to only process the ROIs instead of the full frame.

Figure 16 depicst the operations needed to identify ROIs. The first operation to identify ROIs is to upscale the acoustic heatmap before applying the colormap. On this acoustic heatmap a threshold is applied. All pixels that are below the threshold are set to zero, while pixels above the threshold are set to one. This is done by using the OpenCV function *threshold* [45]. After applying the threshold the contours are extracted and a bounding box for each contour is generated. This is done by using the OpenCV functions *findContours* [46] and *boundingRect* [47]. Timing for detecting different amounts of ROIs can be found in Figure 17. During the experiments, each ROI is 1/16 the size of the image and none of the ROIs overlapped. The time to detect the ROIs increases linearly with the resolution. The time to identify ROIs increases with the number of ROIs, but this increment is lower than when changing the resolution.

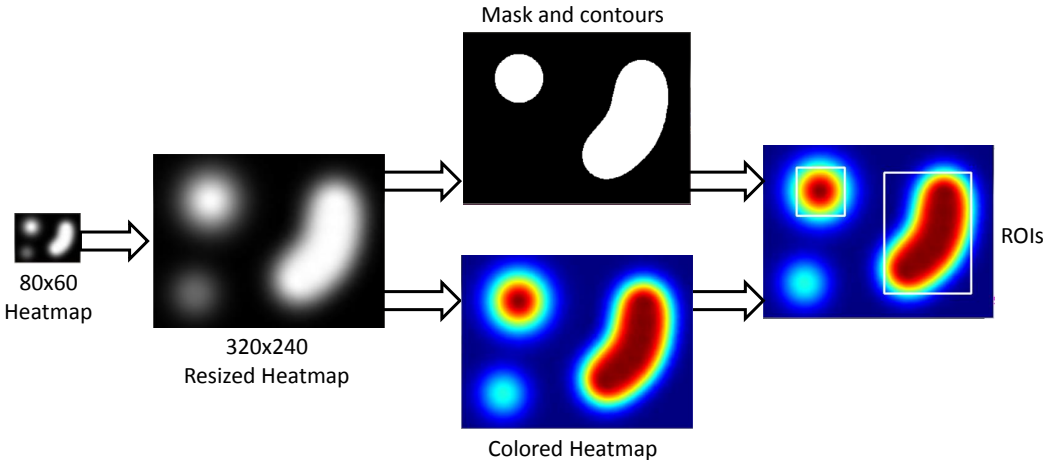

Figure 16. Operations applied to identify Regions-of-Interest (ROIs).

Figure 17. Time for detecting one, two, four or eight ROIs. The gray lines indicate the confidence interval.

5.2.4. RGB Reading from Camera

The acoustic camera combines the acoustic heatmap with an RGB frame from an RGB camera. During the experiments a low cost USB camera is used [48,49]. The camera operates at 30 FPS and has a resolution of 640 × 480. Frames from the camera are read directly on the CPU, and not in the FPGA. This adds the advantage that the camera can be replaced by another model, with a different pixel encoding, resolution or frame rate without the need to make changes to the code or architecture. The openCV function *VideoCapture::read()* is used [50] to read from the camera, which blocks the thread until it has read a frame from the camera. This function also ensures that the frame is provided in RGB format. The current camera uses YUYV encoding, while another may use MJPG at 60 FPS. By reading the frames on the CPU, replacing the camera only requires a physical change of the camera while doing the same when reading the frames in FPGA would also require changes in the FPGA logic.

The duration of the *VideoCapture::read()* method is not fixed, and varies based on the time between two calls to the read function. This is because of a similar principle as described in Section 4.1. When a second call to the read function happens before a frame is ready, the function will block the thread until the frame is ready. This results in the same behavior for the timing as in Section 4.1 and can be seen in Figure 18. Because there is no buffer that can overflow, there is no need for a second thread. Due to the fact that the read function always waits on a new frame from the camera, the frame rate of the application will always be below the frame rate of the camera.

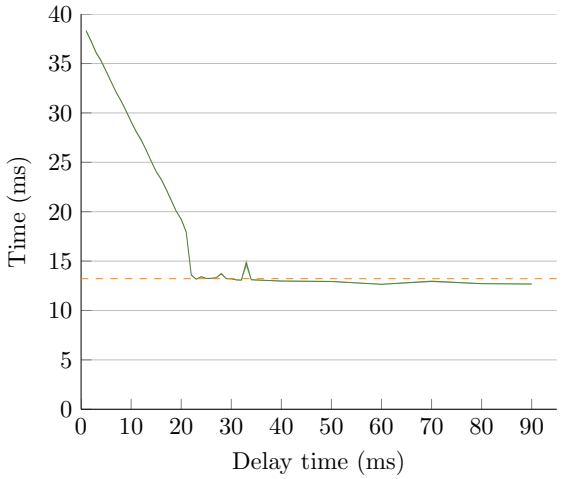

Figure 18. Time for reading a frame from the RGB camera. "usleep" is used for the delay time. The read time (blue line) drops to a minimum when the delay is above 22 ms. The orange line is the average of the read times (13.22 ms) when the delay is more than this 22 ms.

5.2.5. RGB Scaling

The combination of the acoustic heatmap with the RGB frame demands a rescaling of the RGB frame from 640 × 480 to match the upscaled resolution of the acoustic heatmap. Unlike the rescaling of the acoustic heatmap, the frame needs to be downscaled instead of upscaled, resulting in different timings that can be found in Figure 19. Downscaling the frame to 320 × 240 with *Bilinear* takes almost the same time as downscaling it to 160 × 120. The reason is that OpenCV internally uses a special function for downscaling by a factor of two with *Bilinear* specifically aimed to improve the speed for this special case of *Bilinear* interpolation [51]. However, the other two modes follow the same pattern as in the case of the resizing the acoustic heatmap: *Bicubic* is slower than *Bilinear* and *Lanczos* is more than twice as slow as *Bicubic*.

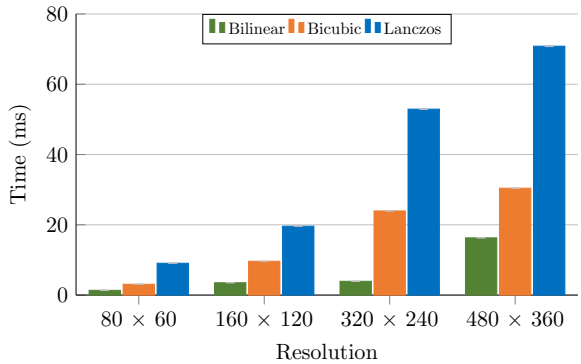

Figure 19. Time for downscaling the RGB to different resolutions with different modes. The original resolution of the image is 640 × 480. The gray lines indicate the confidence interval.

5.2.6. Canny Edge

One of the optional modes needs to apply edge detection on the RGB frame for which the timing can be found in Table 6. This mode uses Canny Edge, which is supported in OpenCV by applying two consecutive functions: *blur* [52] and *canny* [53]. In order to apply these functions, the frame from the RGB camera needs to be converted to a grayscale image first using the *cvtColor* function [54]. Blur reduces the noise in the image and *canny* is used for the edge detection. Like coloring of the acoustic heatmap, the relationship between the resolution and the time to apply canny edge is linear. Furthermore it can also be seen that the canny edge for a resolution of 640 $\times$ 480 takes more than 40 ms. This means that this mode can never be executed in real time for this resolution (remain above 30 FPS, as explained in Section 5.3).

5.2.7. Overlay

After the pre-processing of both, the RGB and acoustic heatmap image can be combined to form one image so it can later be send over wireless or stored in memory. Overlaying both images is done using the OpenCV function *addWeighted* [55]. This is again an operation where the time to perform the operation is linear with the number of pixels. The timing can be found in Table 6. For lower resolutions, the merging is faster than generating the colormap. When the resolution increases, the roles are reversed, and merging the images together becomes slower than generating the colormap. Both functions operate on each pixel individually and do not use neighbouring pixels to determine the color of the output pixel. But the merging needs to process three times more values, because each pixel is represented by an RGB value (24 bits), while the colormap is converting a single channel (8 bit) image into an RGB image. The merging also needs to read a value from two different images and as a result of this, it has to process six times more bytes than the colormap function.

5.2.8. Compression

In order to send the image it first needs to be compressed. The compression is done using the OpenCV functions *imencode* for memory and *imwrite* for writing to a file [56]. Two compression techniques are used. The first one being JPG which is a lossy compression and the second one is Portable Network Graphics (PNG), which is lossless allowing to retrieve the original image. Depending on the mode of the camera the unscaled raw acoustic heatmap is compressed (mode 1) or the combined image of the acoustic heatmap and the RGB camera is compressed (mode 3, 4 or 5). The first has only 8 bits/pixel (single channel) while the latter has 24 bits/pixel (three channels). As a result, there are two different timings and compression sizes. The compression time and size of the compressed image for mode 1 can be found in Figure 20a,c, while the size and timing for compressing the output image generated by mode 3, 4 and 5 can be found in Figure 20b,d.

The time to compress the image using JPG and PNG together with the size of the compressed image are depicted in Figure 20. To measure the time to compress an image, first 1000 acoustic images are generated and stored as single channel images without rescaling. The images are stored as PNG files so that the original image can be reconstructed later and reused for all resolutions. Different images are loaded and rescaled to several resolutions to time the compression techniques. If the RGB compression mode is tested, than the images are converted from grayscale to RGB by applying the colormap.

Because there is no added value to scale the acoustic heatmap and than compress it, the resolutions for the grayscale acoustic heatmap are lower and follow the resolutions of the FPGA. This combined with the fact that only one channel instead of three needs to be compressed, makes this mode faster than the RGB version. It is also clear that it takes more time to store the image on the Secure Digital (SD) card than storing it in memory. This is true for both PNG and JPG. Notice that JPG has a smaller size and takes less time to compress than PNG for the same image size.

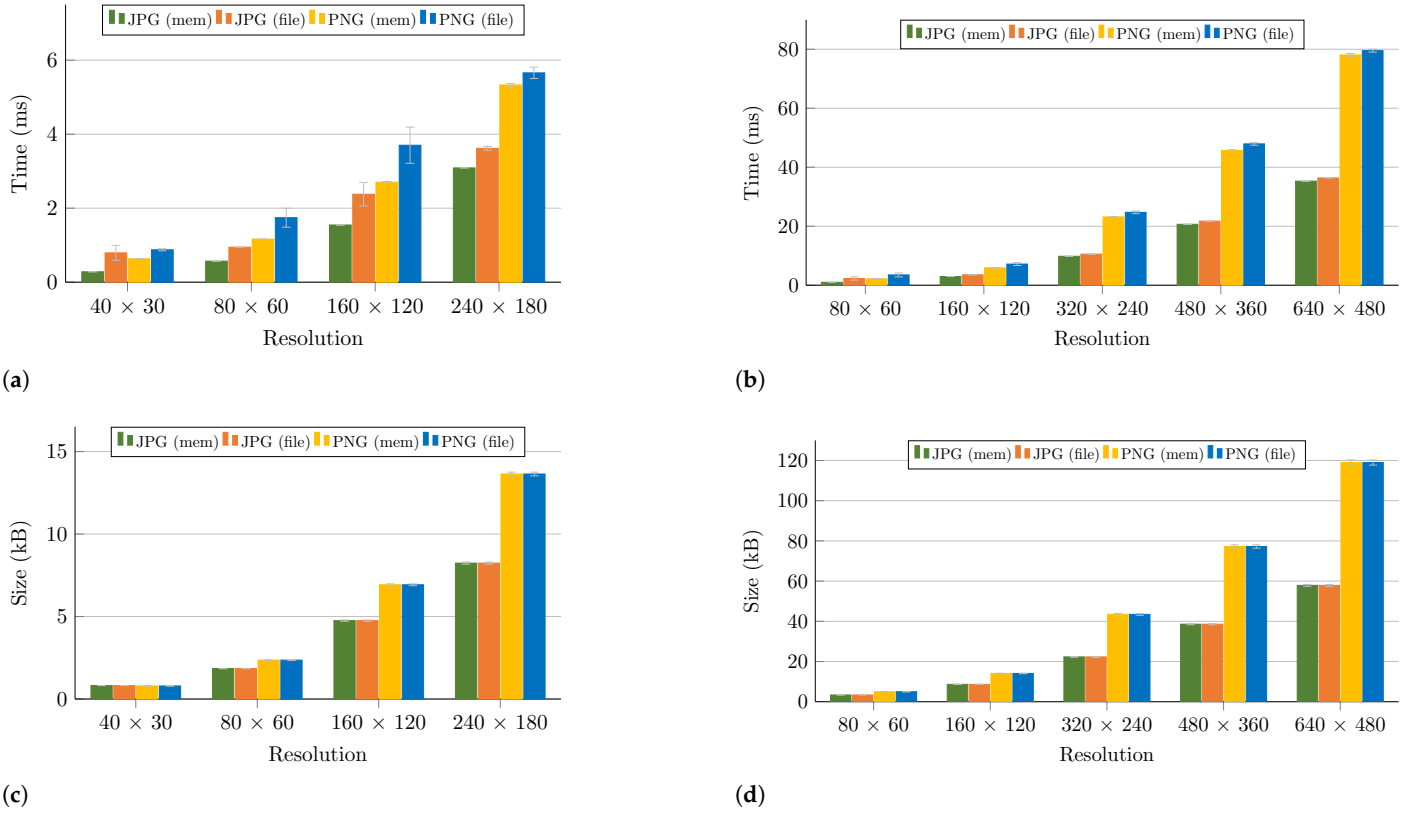

Figure 20. (**a**) Time to compress a grayscale acoustic heatmap for different resolutions and (**c**) the corresponding size of the compressed image (mode 1). (**b**) Time to compress an RGB image for different resolutions and (**d**) the size of the compressed image (mode 3, 4 and 5). The gray lines indicate the confidence interval.

5.3. Analysis of the Back-End: Operational Modes

Using the timings from the previous section, the modes from Section 3.4.1 can be evaluated to identify which modes can operate in real time. The timings for each operational mode, output mode, and resolution can be found in Table 7. Mode 1 does not use scaling and only the four resolutions of the FPGA that have an FPS above 30 are supported. For the other modes, five different resolutions for the CPU are evaluated, the lowest being 80 × 60 and the highest being 640 × 480. Because these resolutions do not match the resolution of the FPGA, the heatmaps are first scaled by a factor of two using *Bilinear*. *Bilinear* is used because it is the fastest. For the resolution of 640 × 480, a resolution on the FPGA of 160 × 240 is used, and the heatmap is scaled by a factor of four. This is because the resolution of 320 × 240 has a FPS of 15.3 on the FPGA, which is below 30 FPS. A frame rate of 30 FPS is considered the minimum frame rate to be real time. This does not mean that the CPU cannot reach 30 FPS if the FPGA works with a resolution of 320 × 240. Like discussed in Section 5.2.1, the time to scale the heatmap depends mainly on the targeted resolution and not on the original resolution. All five modes, together with the timing for each operation and resolution can be found in Table 7.

Because Mode 1 does not need to perform any manipulation of the data, its timing depends only on the chosen output mode. The WSN mode requires compression and sending. It can be seen that sending the data takes the majority of the time for the output mode. This is caused by the fact that the data has to be transferred over UART to the BLE dongle. UART has a much lower BW than BLE and as a result of this it forms a bottleneck in the timing. Replacing the UART by another communication protocol to access the dongle, or choosing a platform that already provides a built-in BLE module would improve the throughput.

The other modes allow the use of the display output mode, and the frame rates can be found in Figure 21. It can be seen that the two lowest resolutions support all 4 modes while achieving real time. On he other hand, the resolution of 640 × 480 is never real time despite the fact that it does not require the RGB frame to be rescaled. When looking at Table 7, one can see that displaying the frame requires 23 ms. This is the minimum time to display the image using the OpenCV function *imshow* [57] combined with *waitkey* [58] because *waitkey* allows the insertion of a wait time, which is set to a minimum of 1 ms. When looking at Figure 21 it can also be seen that the resolution of 480 × 360 is almost real time for mode 2, achieving a frame rate of 29 FPS.

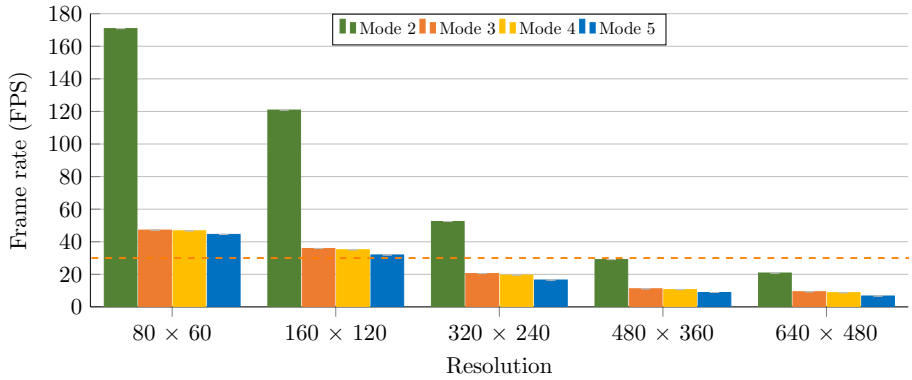

Figure 21. Output frame rate of each mode. The orange line represents the real time threshold of 30 Frames Per Second (FPS). The gray lines indicate the confidence interval.

In the previous sections, the timing of each individual operation is analysed. Figure 22 reflects the timing for the different modes, showing which one are the most time demanding operations. The read time of the RGB camera dominates the time consuming for the lowest resolutions. This is a fixed time independent of the resolution or the operational mode, and can only be changed by replacing the RGB camera. Although the coloring of the acoustic heatmap increases when increasing the output resolution, it is the merging of the two frames which dominates the execution time for the modes 3 and 4, while the edge detection dominates in Mode 5. At 320 × 240 merging takes almost the same amount of time as reading the frame from the camera. And for 480 × 360 and 640 × 480 it even becomes the most time consuming operation. Despite the fact that the resolution of 640 × 480 does not require scaling the RGB frame, the time to scale down the frame to 480 × 360 and merge it with the heatmap is less than merging both frames at 640 × 480.

Table 7. Timing and confidence interval for each operation performed in each mode for different resolutions. The resolution after scaling is indicated in bold. All values are in ms.

Res FPGA	Res CPU	Scaling	HM Color	RGB Read	RGB Scaling	ROI	Canny	Merging	Display		Total Time
									Compression + Storage		
									Compression	Sending	
Mode 1: Raw heatmap											
40 × 30	**40 × 30**								-		-
									0.793 ± 0.201		**0.793**
									0.281 ± 0.003	66.532 ± 0.382	**66.813**
80 × 60	**80 × 60**								-		-
									0.947 ± 0.003		**0.947**
									0.569 ± 0.002	154.325 ± 1.148	**154.894**
160 × 120	**160 × 120**								-		-
									2.378 ± 0.319		**2.378**
									1.546 ± 0.004	401.286 ± 3.203	**402.832**
240 × 180	**240 × 180**								-		-
									3.619 ± 0.052		**3.619**
									3.089 ± 0.006	697.661 ± 5.487	**700.750**
Mode 2: Scaling + HM color + Display											
40 × 30	**80 × 60**	0.223 ± 0.002	3.775 ± 0.017						1.855 ± 0.202		**5.853**
									-		**-**
									-	-	**-**
80 × 60	**160 × 120**	0.704 ± 0.011	4.828 ± 0.027						2.742 ± 0.019		**8.274**
									-		**-**
									-	-	**-**
160 × 120	**320 × 240**	2.453 ± 0.021	9.510 ± 0.026						7.110 ± 0.041		**19.073**
									-		**-**
									-	-	**-**
240 × 180	**480 × 360**	3.707 ± 0.032	16.569 ± 0.024						14.124 ± 0.056		**34.400**
									-		**-**
									-	-	**-**
160 × 120	**640 × 480**	6.580 ± 0.047	17.801 ± 0.114						23.658 ± 0.060		**48.039**
									-		**-**
									-	-	**-**
Mode 3: Scaling + HM color + RGB read + RGB scaling + Output mode											
40 × 30	**80 × 60**	0.223 ± 0.002	3.775 ± 0.017	13.218 ± 0.075	1.353 ± 0.006			0.772 ± 0.003	1.855 ± 0.202		**21.196**
									2.276 ± 0.553		**21.617**
									1.000 ± 0.002	283.791 ± 1.955	**304.132**
80 × 60	**160 × 120**	0.704 ± 0.011	4.828 ± 0.027	13.218 ± 0.075	3.505 ± 0.006			2.883 ± 0.003	2.742 ± 0.019		**27.880**
									3.532 ± 0.101		**28.670**
									2.952 ± 0.007	733.093 ± 5.752	**761.183**
160 × 120	**320 × 240**	2.453 ± 0.021	9.510 ± 0.026	13.218 ± 0.075	4.099 ± 0.016			12.422 ± 0.038	7.110 ± 0.041		**48.812**
									10.536 ± 0.034		**52.238**
									9.812 ± 0.017	1893.760 ± 15.502	**1945.274**
240 × 180	**480 × 360**	3.707 ± 0.032	16.569 ± 0.024	13.218 ± 0.075	16.281 ± 0.040			25.905 ± 0.067	14.124 ± 0.056		**89.804**
									21.724 ± 0.131		**97.404**
									20.654 ± 0.030	3276.708 ± 26.344	**3373.042**
160 × 120	**640 × 480**	6.580 ± 0.047	17.801 ± 0.114	13.218 ± 0.075	-			45.961 ± 0.098	23.658 ± 0.060		**107.218**
									36.368 ± 0.067		**119.982**
									35.283 ± 0.050	4917.320 ± 39.874	**5036.136**
Mode 4: Scaling + HM color + RGB read + RGB scaling + ROI + Merging + Output mode											
40 × 30	**80 × 60**	0.223 ± 0.002	3.775 ± 0.017	13.218 ± 0.075	1.353 ± 0.006	0.211 ± 0.002		0.772 ± 0.003	1.855 ± 0.202		**21.407**
									2.276 ± 0.553		**21.828**
									1.000 ± 0.002	283.791 ± 1.955	**304.343**
80 × 60	**160 × 120**	0.704 ± 0.011	4.828 ± 0.027	13.218 ± 0.075	3.505 ± 0.006	0.626 ± 0.003		2.883 ± 0.003	2.742 ± 0.019		**28.506**
									3.532 ± 0.101		**29.296**
									2.952 ± 0.007	733.093 ± 5.752	**761.809**
160 × 120	**320 × 240**	2.453 ± 0.021	9.510 ± 0.026	13.218 ± 0.075	4.099 ± 0.016	2.361 ± 0.005		12.422 ± 0.038	7.110 ± 0.041		**51.173**
									10.536 ± 0.034		**54.599**
									9.812 ± 0.017	1893.760 ± 15.502	**1947.635**
240 × 180	**480 × 360**	3.707 ± 0.032	16.569 ± 0.024	13.218 ± 0.075	16.281 ± 0.040	4.254 ± 0.029		25.905 ± 0.067	14.124 ± 0.056		**94.058**
									21.724 ± 0.131		**101.658**
									20.654 ± 0.030	3276.708 ± 26.344	**3377.296**
160 × 120	**640 × 480**	6.580 ± 0.047	17.801 ± 0.114	13.218 ± 0.075	-	7.314 ± 0.018		45.961 ± 0.098	23.658 ± 0.060		**114.532**
									36.368 ± 0.067		**127.242**
									35.283 ± 0.050	4917.320 ± 39.874	**5043.477**
Mode 5: Scaling + HM color + RGB read + RGB scaling + Canny + Merging + Output mode											
40 × 30	**80 × 60**	0.223 ± 0.002	3.775 ± 0.017	13.218 ± 0.075	1.353 ± 0.006		1.262 ± 0.017	0.772 ± 0.003	1.855 ± 0.202		**22.458**
									2.276 ± 0.553		**22.879**
									1.000 ± 0.002	283.791 ± 1.955	**305.394**
80 × 60	**160 × 120**	0.704 ± 0.011	4.828 ± 0.027	13.218 ± 0.075	3.505 ± 0.006		3.463 ± 0.035	2.883 ± 0.003	2.742 ± 0.019		**31.343**
									3.532 ± 0.101		**32.133**
									2.952 ± 0.007	733.093 ± 5.752	**764.646**
160 × 120	**320 × 240**	2.453 ± 0.021	9.510 ± 0.026	13.218 ± 0.075	4.099 ± 0.016		11.717 ± 0.076	12.422 ± 0.038	7.110 ± 0.041		**60.529**
									10.536 ± 0.034		**63.955**
									9.812 ± 0.017	1893.760 ± 15.502	**1956.991**
240 × 180	**480 × 360**	3.707 ± 0.032	16.569 ± 0.024	13.218 ± 0.075	16.281 ± 0.040		23.274 ± 0.108	25.905 ± 0.067	14.124 ± 0.056		**113.078**
									21.724 ± 0.131		**120.678**
									20.654 ± 0.030	3276.708 ± 26.344	**3396.316**
160 × 120	**640 × 480**	6.580 ± 0.047	17.801 ± 0.114	13.218 ± 0.075	-		41.942 ± 0.146	45.961 ± 0.098	23.658 ± 0.060		**149.160**
									36.368 ± 0.067		**161.870**
									35.283 ± 0.050	4917.320 ± 39.874	**5078.105**

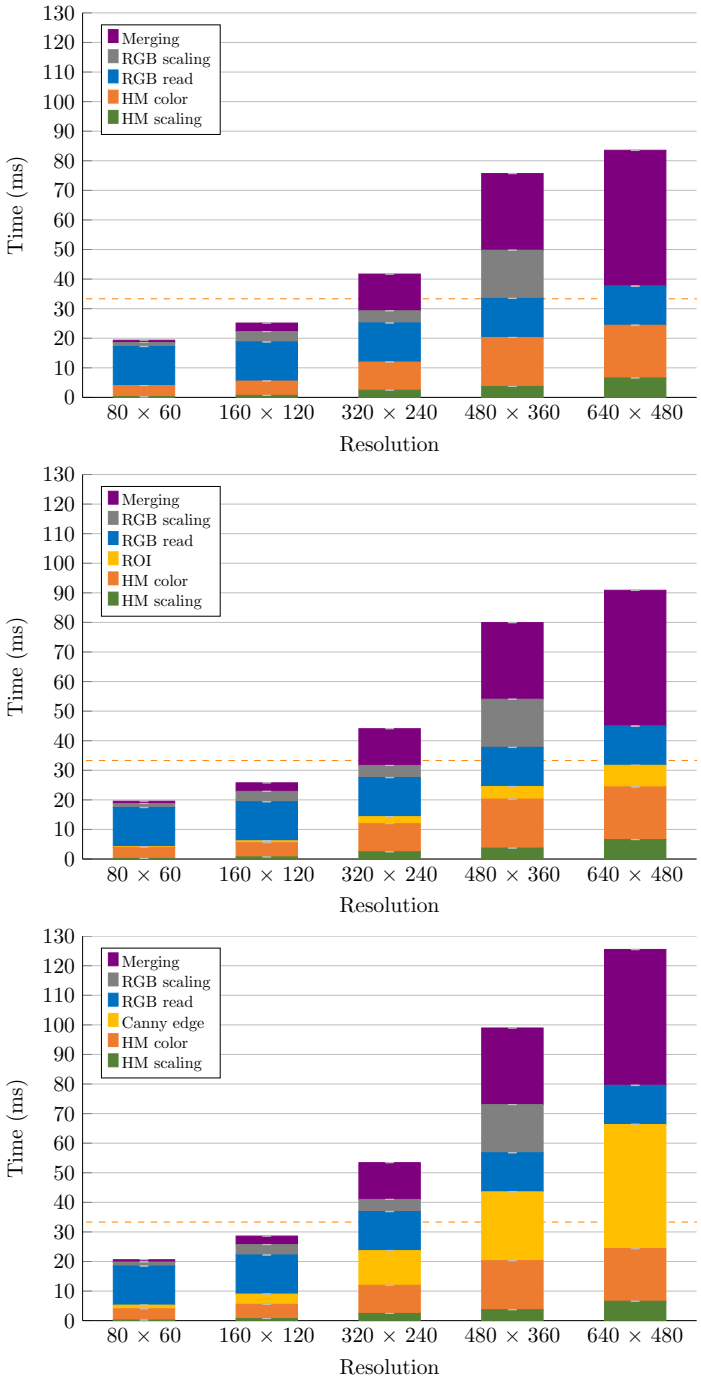

Figure 22. Time for each computational operation in Mode 3 (**left**), Mode 4 (**center**) and Mode 5 (**right**) for different resolutions. The orange line represents the threshold of 33 ms (30 FPS). The gray lines indicate the confidence interval.

Notice that the detection of ROIs in Mode 4 is not a time demanding operation. Therefore, like discussed in Section 5.2.3, the ROI can be used to improve the timing by discarding frames that do not have any ROI or by only processing the ROIs instead of full frames. This is of course only possible when the output mode is not display. The new multithreaded approach accelerates the identification of multiple ROIs. Although the interval between processed frames changes depending on the amount of ROIs that are detected in the acoustic heatmap, there is no queue building up of unprocessed frames between the FPGA and CPU, allowing the FPGA and the CPU to stay synchronised. Without this multithreaded approach, a buffer should be allocated that is big enough to store multiple frames, increasing the amount of resources consumed, or a handshake is

needed between the FPGA and the CPU, decreasing the throughput between FPGA and CPU because the CPU needs to wait for a start signal from the FPGA and make sure that it has read a complete frame, and not two partial frames.

The NORDIC USB BLE dongle achieves the highest throughput when using UART for the communication between the CPU and the BLE dongle. However, UART limits the throughput from 92 to 115.2 kbps while the BLE dongle supports theoretical thoughputs up to 2 Mbps [36]. This theoretical throughput, combined with the average compression size, can be used to estimate achievable timing and throughput. Despite this comparison is only performed for Mode 3, there is an analog relationship with the other modes. A comparison between the current throughput and achievable throughput can be found in Table 8. Notice that the throughput increases for all resolutions with more than a factor of 9. The resolution 80 × 60 has an achievable timing of 33.821 ms, which is almost 30 FPS. One of the advantages of the M3-AC system is its flexibility to support different wireless communication standards, by using additional WSN motes or another communication protocol. For instance, the BLE dongle could be replaced by a wireless protocol that supports a higher BW, e.g., a Wi-Fi dongle or a dongle that supports 4G/5G. Although this solution would support video and audio streaming, it would increase significantly the overall power consumption.

Table 8. Comparison between current timing/throughput and achievable timing/throughput for mode 3. The achievable throughput is based on the maximum throughput of 2 Mbps that is supported by the BLE dongle [36].

Resolution	Current Timing [ms]	Theoretical Timing [ms]	Current Throughput [kB/s]	Theoretical Throughput [kB/s]
80 × 60	304,132	33,821	11,081	99,642
160 × 120	761,183	62,710	11,370	138,016
320 × 240	1,945,274	140,762	11,470	158,509
480 × 360	3,373,042	250,662	11,438	153,920
640 × 480	5,036,163	350,379	11,494	165,204

6. Conclusions

The presented M3-AC system is designed to not only offering wireless capabilities but also multiple operational modes to satisfy different applications. The embedded architecture is designed to exploit the FPGAs features by making use of the data parallelism and operating in pipeline to achieve high FPS for relatively high resolutions. In fact, the optimized architecture not only almost doubles the performance of the original one, the relatively low resource consumption also enables the use of larger microphone arrays composed of more than 50 microphones at a very low power consumption. On the other hand, the multithread approach allows a better workload balance between the FPGA and the CPU. The flexibility of the CPU facilitates the support of multiple modes with different resolutions. By profiling the timing of the different operations performed in each mode, the resolution of the FPGA can be adapted to match the timing of the CPU. Thanks to adapting the resolution, it is possible to remain in real time, even with the more time-demanding modes. As a consequence, the M3-AC system achieves real time performance in several supported modes providing multiple configurations to satisfy the constrained bandwidth.

Author Contributions: Methodology, J.V. and B.d.S.; hardware-software co-design, J.V., L.L. and B.d.S.; validation, J.V. and B.d.S.; writing—original draft preparation, J.V., B.d.S. and A.B.; writing—review and editing, J.V., B.d.S. and A.B.; supervision, B.d.S., A.B. and A.T.; funding acquisition, A.T.; All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially supported by the European Regional Development Fund (ERDF) and the Brussels-Capital Region-Innoviris within the framework of the Operational Programme 2014–2020 through the ERDF-2020 Project ICITYRDI.BRU. This work is also part of the COllective Research NETworking (CORNET) project *"AITIA: Embedded AI Techniques for Industrial Applications"* [59].

The Belgian partners are funded by VLAIO under grant number HBC.2018.0491, while the German partners are funded by the BMWi (Federal Ministry for Economic Affairs and Energy) under IGF-Project Number 249 EBG.

Data Availability Statement: The data used to support the findings of this study are available from the corresponding authors upon request.

Acknowledgments: The authors would like to thanks Xilinx for the provided software and hardware under the Xilinx University Program (XUP) donation.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Tiete, J.; Domínguez, F.; da Silva, B.; Segers, L.; Steenhaut, K.; Touhafi, A. SoundCompass: A Distributed MEMS Microphone Array-Based Sensor for Sound Source Localization. *Sensors* **2014**, *14*, 1918–1949. [CrossRef] [PubMed]
2. Da Silva, B.; Segers, L.; Rasschaert, Y.; Quevy, Q.; Braeken, A.; Touhafi, A. A Multimode SoC FPGA-Based Acoustic Camera for Wireless Sensor Networks. In Proceedings of the 13th International Symposium on Reconfigurable Communication-Centric Systems-on-Chip, ReCoSoC 2018, Lille, France, 9–11 July 2018; pp. 1–8.
3. Zimmermann, B.; Studer, C. FPGA-based real-time acoustic camera prototype. In Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS), Paris, France, 30 May–2 June 2010; p. 1419.
4. Perrodin, F.; Nikolic, J.; Busset, J.; Siegwart, R. Design and calibration of large microphone arrays for robotic applications. In Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vilamoura-Algarve, Portugal, 7–12 October 2012; pp. 4596–4601.
5. Sanchez-Hevia, H.; Gil-Pita, R.; Rosa-Zurera, M.; others. FPGA-based real-time acoustic camera using PDM MEMS microphones with a custom demodulation filter. In Proceedings of the Sensor Array and Multichannel Signal Processing Workshop (SAM), A Coruna, Spain, 22–25 June 2014; pp. 181–184.
6. Sánchez-Hevia, H.A.; Mohino-Herranz, I.; Gil-Pita, R.; Rosa-Zurera, M. Memory Requirements Reduction Technique for Delay Storage in Real Time Acoustic Cameras. In Proceedings of the Audio Engineering Society Convention 136, Audio Engineering Society, Berlin, Germany, 26–29 April 2014.
7. Kim, Y.; Kang, J.; Lee, M. Developing beam-forming devices to detect squeak and rattle sources by using FPGA. INTER-NOISE and NOISE-CON Congress and Conference Proceedings. *Inst. Noise Control Eng.* **2014**, *249*, 4582–4587.
8. Netti, A.; Diodati, G.; Camastra, F.; Quaranta, V. FPGA implementation of a real-time filter and sum beamformer for acoustic antenna. INTER-NOISE and NOISE-CON Congress and Conference Proceedings. *Inst. Noise Control Eng.* **2015**, *250*, 3458–3469.
9. Bourgeois, J.; Minker, W. (Eds.) Linearly Constrained Minimum Variance Beamforming. In *Time-Domain Beamforming and Blind Source Separation: Speech Input in the Car Environment*; Springer: Boston, MA, USA, 2009; pp. 27–38.
10. Izquierdo, A.; Villacorta, J.J.; del Val Puente, L.; Suárez, L. Design and evaluation of a scalable and reconfigurable multi-platform system for acoustic imaging. *Sensors* **2016**, *16*, 1671. [CrossRef] [PubMed]
11. Del Val, L.; Izquierdo, A.; Villacorta, J.J.; Suárez, L. Using a Planar Array of MEMS Microphones to Obtain Acoustic Images of a Fan Matrix. *J. Sens.* **2017**, *2017*, 3209142. [CrossRef]
12. Izquierdo, A.; Villacorta, J.J.; del Val, L.; Suárez, L.; Suárez, D. Implementation of a Virtual Microphone Array to Obtain High Resolution Acoustic Images. *Sensors* **2017**, *18*, 25. [CrossRef] [PubMed]
13. Seo, S.W.; Kim, M. 3D Impulsive Sound-Source Localization Method through a 2D MEMS Microphone Array using Delay-and-Sum Beamforming. In Proceedings of the 9th International Conference on Signal Processing Systems, Auckland, New Zealand, 27–30 November 2017; pp. 170–174.
14. Seo, S.W.; Kim, M. Estimation of 3D ball motion using an infrared and acoustic vector sensor. In Proceedings of the 2017 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea, 18–20 October 2017; pp. 1047–1049.
15. Seo, S.W.; Yun, S.; Kim, M.G.; Sung, M.; Kim, Y. Screen-Based Sports Simulation Using Acoustic Source Localization. *Appl. Sci.* **2019**, *9*, 2970. [CrossRef]
16. Fréchette-Viens, J.; Quaegebeur, N.; Atalla, N. A Low-Latency Acoustic camera for Transient Noise Source Localization. In Proceedings of the 8th Berlin Beamforming Conference, Berlin, Germany, 2–3 March 2020.
17. Costas, L.; Fernández-Molanes, R.; Rodríguez-Andina, J.J.; Fariña, J. Characterization of FPGA-master ARM communication delays in zynq devices. In Proceedings of the 2017 IEEE International Conference on Industrial Technology (ICIT), Toronto, ON, Canada, 22–25 March 2017; pp. 942–947.
18. Lin, Z.; Chow, P. Zcluster: A zynq-based hadoop cluster. In Proceedings of the 2013 International Conference on Field-Programmable Technology (FPT), Kyoto, Japan, 9–11 December 2013; pp. 450–453.
19. Xillybus Host Application Programming Guide for Linux. Available online: http://xillybus.com/downloads/doc/xillybus_host_programming_guide_linux.pdf (accessed on 27 October 2020).
20. Xilinx OpenCV User Guide, UG1233 (v2019.1). Available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1233-xilinx-opencv-user-guide.pdf (accessed on 12 November 2020).

21. OpenCV Modules . Available online: https://docs.opencv.org/4.4.0/ (accessed on 12 November 2020).
22. Xilinx Zynq–7000 All Programmable Soc Overview. Available online: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (accessed on 15 January 2021).
23. Knowles Acoustics MEMS Microphones SPH0644HM4H-1 RevB Datasheet. Available online: https://www.digikey.jp/htmldatasheets/production/3083460/0/0/1/sph0644hm4h-1-datasheet-.html (accessed on 15 September 2020).
24. Analog Devices, ADMP521 Datasheet. Available online: http://www.analog.com/media/en/technical-documentation/obsolete-data-sheets/ADMP521.pdf (accessed on 15 September 2020).
25. Da Silva, B.; Braeken, A.; Steenhaut, K.; Touhafi, A. Design Considerations When Accelerating an FPGA-Based Digital Microphone Array for Sound-Source Localization. *J. Sens.* **2017**, *2017*, 20. [CrossRef]
26. Da Silva, B.; Braeken, A.; Touhafi, A. FPGA-Based Architectures for Acoustic Beamforming with Microphone Arrays: Trends, Challenges and Research Opportunities. *Computers* **2018**, *7*, 29. [CrossRef]
27. Da Silva, B.; Segers, L.; Braeken, A.; Steenhaut, K.; Touhafi, A. Design Exploration and Performance Strategies Towards Power-Efficient FPGA-based Architectures for Sound Source Localization. *J. Sens.* **2019**, *2019*, 31. [CrossRef]
28. Da Silva, B.; Segers, L.; Braeken, A.; Touhafi, A. Runtime reconfigurable beamforming architecture for real-time sound-source localization. In Proceedings of the 26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, 29 August–2 September 2016; pp. 1–4.
29. Da Silva, B.; Segers, L.; Braeken, A.; Steenhaut, K.; Touhafi, A. A Low-Power FPGA-Based Architecture for Microphone Arrays in Wireless Sensor Networks. In Proceedings of the Applied Reconfigurable Computing. Architectures, Tools, and Applications—14th International Symposium, ARC 2018, Santorini, Greece, 2–4 May 2018; pp. 281–293.
30. Saff, E.B.; Kuijlaars, A.B. Distributing many points on a sphere. *Math. Intell.* **1997**, *19*, 5–11. [CrossRef]
31. Taghizadeh, M.J.; Garner, P.N.; Bourlard, H. Microphone array beampattern characterization for hands-free speech applications. In Proceedings of the 2012 IEEE 7th Sensor Array and Multichannel Signal Processing Workshop (SAM), Hoboken, NJ, USA, 17–20 June 2012; pp. 465–468.
32. Johnson, D.H.; Dudgeon, D.E. *Array Signal Processing: Concepts and Techniques*, 1st ed.; Pearson: London, UK, 1993.
33. Segers, L.; Vandendriessche, J.; Vandervelden, T.; Lapauw, B.J.; da Silva, B.; Braeken, A.; Touhafi, A. CABE: A Cloud-Based Acoustic Beamforming Emulator for FPGA-Based Sound Source Localization. *Sensors* **2019**, *19*, 3906. [CrossRef] [PubMed]
34. Xillybus. Available online: http://xillybus.com (accessed on 15 September 2020).
35. OpenCV. Available online: https://opencv.org (accessed on 15 September 2020).
36. nRF52840 Dongle. Available online: https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF52840-Dongle-product-brief.pdf (accessed on 15 September 2020).
37. Bluetooth Low Energy 5.0, Maximum Throughput. Available online: https://www.novelbits.io/bluetooth-5-speed-maximum-throughput/ (accessed on 26 October 2020).
38. Bluetooth-SIG. Specification of the Bluetooth system. *Bluetooth* **2010**, *6*, 14–24.
39. GCC Linux Manual. Available online: https://man7.org/linux/man-pages/man1/gcc.1.html (accessed on 19 August 2020).
40. Ye, Z.; Suri, J.; Sun, Y.; Janer, R. Four image interpolation techniques for ultrasound breast phantom data acquired using Fischer's full field digital mammography and ultrasound system (FFDMUS): A comparative approach. In Proceedings of the IEEE International Conference on Image Processing 2005, Genova, Italy, 14 September 2005; Volume 2.
41. Sharma, H.; Saurav, S.; Singh, S.; Saini, A.K.; Saini, R. Analyzing impact of image scaling algorithms on viola-jones face detection framework. In Proceedings of the 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Kochi, India, 10–13 August 2015; pp. 1715–1718.
42. OpenCV Resize. Available online: https://docs.opencv.org/4.4.0/da/d54/group__imgproc__transform.html#ga47a974309e9102f5f08231edc7e7529d (accessed on 14 August 2020).
43. OpenCV ApplyColorMap. Available online: https://docs.opencv.org/4.4.0/d3/d50/group__imgproc__colormap.html#gadf478a5e5ff49d8aa24e726ea6f65d15 (accessed on 14 August 2020).
44. OpenCV: How to Use the OpenCV Parallel_for_ to Parallelize Your Code . Available online: https://docs.opencv.org/4.4.0/d7/dff/tutorial_how_to_use_OpenCV_parallel_for_.html (accessed on 14 August 2020).
45. OpenCV Threshold. Available online: https://docs.opencv.org/4.4.0/d7/d1b/group__imgproc__misc.html#gae8a4a146d1ca78c626a53577199e9c57 (accessed on 14 August 2020).
46. OpenCV FindContours. Available online: https://docs.opencv.org/4.4.0/d3/dc0/group__imgproc__shape.html#gadf1ad6a0b82947fa1fe3c3d497f260e0 (accessed on 2 September 2020).
47. OpenCV BoundingRect. Available online: https://docs.opencv.org/4.4.0/d3/dc0/group__imgproc__shape.html#ga103fcbda2f540f3ef1c042d6a9b35ac7 (accessed on 2 September 2020).
48. Robu.in. Available online: https://robu.in/product/1-4-cmos-640x480-usb-camera-with-collapsible-cable-for-raspberry-pi-3 (accessed on 7 September 2020).
49. Banggood. Available online: https://www.banggood.com/Raspberry-Pi-USB-Camera-Module-with-Adjustable-Focusing-Range-for-Raspberry-Pi-32BB-p-1462594.html?cur_warehouse=CN (accessed on 7 September 2020).
50. OpenCV VideoCapture. Available online: https://docs.opencv.org/4.4.0/d8/dfe/classcv_1_1VideoCapture.html#a473055e77dd7faa4d26d686226b292c1 (accessed on 14 August 2020).

51. OpenCV Resize Source Code (Github). Available online: https://github.com/opencv/opencv/blob/61c4cfd89624617121725a119482dde278f82955/modules/imgproc/src/resize.cpp#L3730 (accessed on 4 September 2020).
52. OpenCV Blur. Available online: https://docs.opencv.org/4.4.0/d4/d86/group__imgproc__filter.html#ga8c45db9afe636703801b0b2e440fce37l (accessed on 14 August 2020).
53. OpenCV Canny. Available online: https://docs.opencv.org/4.4.0/dd/d1a/group__imgproc__feature.html#ga04723e007ed888ddf11d9ba04e2232de (accessed on 14 August 2020).
54. OpenCV CvtColor. Available online: https://docs.opencv.org/4.4.0/d8/d01/group__imgproc__color__conversions.html#ga397ae87e1288a81d2363b61574eb8cab (accessed on 12 November 2020).
55. OpenCV AddWeighted. Available online: https://docs.opencv.org/4.4.0/d2/de8/group__core__array.html#gafafb2513349db3bcff51f54ee5592a19 (accessed on 14 August 2020).
56. OpenCV Image Encoding. Available online: https://docs.opencv.org/4.4.0/d4/da8/group__imgcodecs.html (accessed on 14 August 2020).
57. OpenCV Imshow. Available online: https://docs.opencv.org/4.4.0/d7/dfc/group__highgui.html#ga453d42fe4cb60e5723281a89973ee563 (accessed on 10 September 2020).
58. OpenCV Waitkey. Available online: https://docs.opencv.org/4.4.0/d7/dfc/group__highgui.html#ga5628525ad33f52eab17feebcfba38bd7 (accessed on 10 September 2020).
59. Brandalero, M.; Ali, M.; Le Jeune, L.; Hernandez, H.G.M.; Veleski, M.; da Silva, B.; Lemeire, J.; Van Beeck, K.; Touhafi, A.; Goedemé, T.; et al. AITIA: Embedded AI Techniques for Embedded Industrial Applications. In Proceedings of the 22020 International Conference on Omni-layer Intelligent Systems (COINS), Barcelona, Spain, 31 August–2 September 2020; pp. 1–7.