2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

FPGA-based Embedded System Implementation of Audio Signal Alignment

Luca Stornaiuolo, Massimo Perini, Marco D. Santambrogio, Donatella Sciuto
Politecnico di Milano, Dipartimento di Elettronica Informazione e Bioingegneria (DEIB), Milan, Italy
{luca.stornaiuolo, marco.santambrogio, donatella.sciuto}@polimi.it
massimo.perini@mail.polimi.it

***Abstract*—FPGAs are considered a valuable solution for embedded system applications thanks to their performance, energy efficiency and capability to face system failures. However, the number of available applications is limited due to the learning curve needed to customize FPGA-based accelerators. As proof of this, Xilinx recently released PYNQ, a platform for Zynq SoC that relies on Python and overlays to ease the integration of functionalities of the programmable logic into applications.**

In this work, we build upon this framework to implement an optimized embedded design for audio alignment and we integrated it in the Python applications workflow. In particular, we provide a custom accelerator designed for PYNQ and the software interface to transparently exploit the programmable logic from the Python code runs on the embedded CPU. We then compare the executions on two different devices: the PYNQ-Z1 and the Raspberry Pi 3. Our FPGA accelerated implementation is able to reach a speedup of 12.4x with respect to the PYNQ-Z1, when only the CPU is used, and a speedup of 5.5x with respect to the Raspberry Pi 3 version.

***Index Terms*—Zynq, SoC, PYNQ, Python, NumPy, FPGA, Audio Alignment**

I. Introduction

FPGAs are experiencing an exceptionally favorable moment, as demonstrated by Intel's acquisition of Altera, Microsoft's Catapult project [1], and Amazon's integration of FPGAs as accelerators in their cloud offerings[1]. The slowing of Moore's law, and the rise of fields as artificial intelligence and computational biology, are indeed shifting the interest of industry and academia towards less conventional computing architectures, that can meet the ever increasing demand for performance and energy efficiency, an exemplary choice being precisely FPGAs. For this reason, FPGAs market is expected to reach $12.1 Billion by 2024[2], showing a Compound Annual Growth Rate (CAGR) of 7.3% starting from 2016.

However, no matter how much FPGA technology has matured, the usability barrier is still preventing the mainstream adoption from happening. As a matter of fact, integrating FPGA-based hardware accelerators into applications today is still a cumbersome experience. The current implementation flow requires specific skills and knowledge of low-level tools that are simply out of reach for the largest part of software developers, and albeit High Level Synthesis (HLS) does mitigate some difficulties, by at least offering the possibility to use higher level languages, today's available tools still require to go through the same development process.

In an effort to address the usability challenge, Xilinx recently released the PYNQ (PYthon productivity for zyNQ) platform [2]. Zynq architecture integrates a multi-core ARM processor with an FPGA into a single chip. With Python [3], developers can build complex applications very quickly, by leveraging its high level of abstraction and the plethora of available libraries. PYNQ then offers the possibility to exploit the programmable logic within the Python environment by means of *overlays*, or *hardware libraries*. These overlays are essentially FPGA designs whose functionalities are made available to the user as Python Application Programming Interface (API). Developers can then simply import and use these libraries, exploiting the programmable logic while staying at the pure-software level.

Within this paper, we present our work on accelerating Audio Signal Alignment for a PYNQ-based embedded system. The Audio Signal Alignment application we target is mainly based on two scientific functions: the Cross-Correlation and the Fast Fourier Transform. Since we deal with the Python environment, we started from the *NumPy* (i.e. the most used Python library for scientific calculus[3]) implementations of these two functions and we accelerate them by offloading part of the computation from the processing system to the programmable logic available on Zynq SoCs. We exploited the PYNQ overlay concept to build a hardware library that can be integrated into the Xilinx platform and can be used transparently by the end users. We have done this, so that software developers and data scientists can exploit the accelerated version of the functions by simply changing the import of *NumPy* in their Data Science applications, with everything handled automatically from the system.

We have chosen Audio Signal Alignment also taking into consideration the scenario where part of the scientific computation and signal preprocessing is performed on an embedded device before sending the data to the cloud servers. This reflects the Fog Computing paradigm [4], which, nowadays, is increasingly gaining ground. For this reason, we have compared the results on the PYNQ-Z1 board with the same application performed on a Raspberry Pi device, one of the most widespread solutions for embedded systems.

[1]https://aws.amazon.com/it/ec2/instance-types/f1

[2]https://www.variantmarketresearch.com/report-categories/semiconductor-electronics/field-programmable-gate-array-market

[3]http://www.numpy.org

978-1-5386-5555-9/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPSW.2019.00031

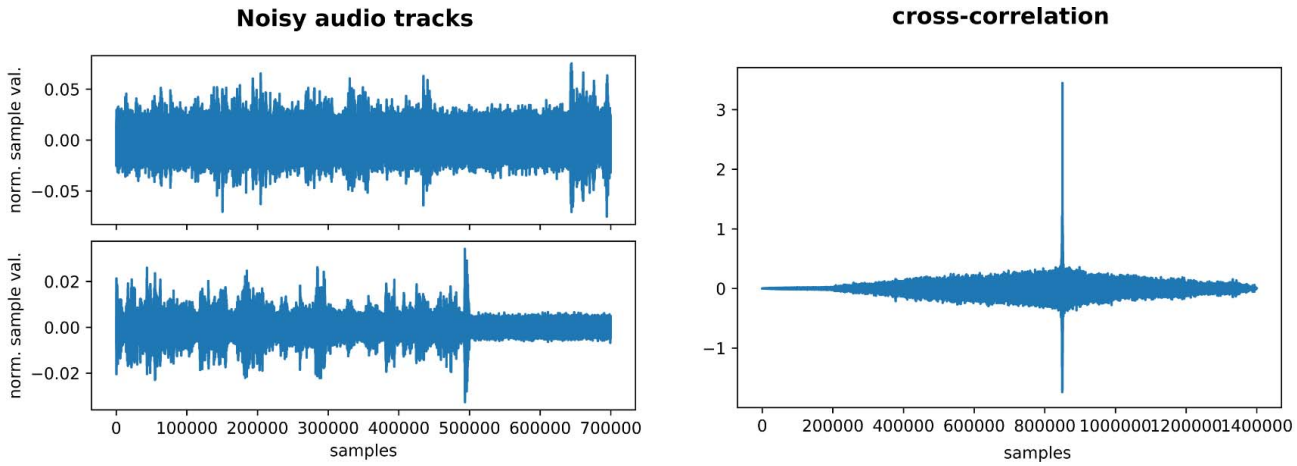

Figure 1. These two figures describe the Audio Signal Alignment application. The left image shows two noisy audio of the same conference recorded with different microphones in different places. Common recordings is in the range [150000:650000] of the first track and in the range [0:500000] of the second track. The right image shows the Cross Correlation function result. The peak of the Cross Correlation function is in the 849946^{th} sample. The middle point is in the 699999^{th} sample so we need to remove 149947 samples from the first signal in order to have them synchronized (with an error of 53 samples).

II. Background and Motivation

In this section we provide a description of the Audio Signal Alignment application, we briefly present the PYNQ platform with the explanation of the rationale behind overlays, then we show an overview about the embedded devices employed in this project before ending the section with the related work found in the literature.

A. Audio Signal Alignment

Signal alignment is a real-world problem suitable for embedded system devices and in which our approach can be adopted. In these kinds of problems, we have to align two or more misaligned noise-corrupted signals in which the same event has been recorded. These kinds of issues can occur in many areas including biomedical and audio field and therefore we studied how hardware acceleration can be applied to the latter case. In the audio field, we often have multiple signals of the same event recorded by different sources. There are many reasons why this approach is adopted, such as the fact of being able to generate a less noisy signal. Since it is not always possible to have recordings synchronized together, an embedded system able to synchronize audio tracks and remove the portion of the signal not related to the event we are interested in can be useful. An example can be founded in the scenario of multiple audio signals of a recorded conference. We may have many reasons to process them, e.g. speech-recognition, but the audio sources could be recorded by multiple microphones with no common starting or ending points. These signals could also have multiple different sources of noise (e.g. crosstalk). In this case, we may want an embedded device in charge of sending to the server only the common parts of the different signals adopting a Fog Computing paradigm, i.e. avoid sending useless data that can be removed during pre-processing.

A similar approach can also be adapted to be able to synchronize multiple video streams using their audio channels. In this case, the saved data transfer will be even higher. The software implementation of the audio alignment application is inspired by [5]. Specifically, given two signals, this algorithm requires to compute the Cross-Correlation function of their samples and find the value x in which the Cross-Correlation function is maximized. Defining $N = len(first\_signal)$ and $M = len(second\_signal)$, $x - \frac{(N+M-1)}{2}$ is the number of samples that we need to remove from the first signal in order to align it with the second one. In the case of negative value, these samples need to be removed from the second signal.

In order to improve the performances of the algorithm, we furthermore extracted a sequence of MFCC feature vectors from both signals using [6]. This step requires applying the Fast Fourier Transform of the signals. We then computed the Cross-Correlation function of the feature matrices along the time axis and calculated the average of the points in which the Cross-Correlation function is maximized. Finally, the output of the cross-correlated signals is averaged with the output of the cross-correlated feature matrices to reduce the percentage error of alignment. Figure 1 shows the result of applying Cross Correlation function to two audio signals recorded by different sources.

B. The PYNQ Platform

PYNQ is a Xilinx platform that targets Zynq SoCs, and its objective is to allow developers to write applications that exploit the programmable logic without having to use the low-level design tools needed to design programmable logic circuits. PYNQ relies on Python as the *productivity language* of choice, a decision driven by its incredible popularity [3], and the fact that Python raises the level of programming abstraction which results in more concise, expressive code, that is in turn less prone to errors and faster to write. Moreover, PYNQ uses CPython, the default and most used Python interpreter, that is written in C and comes with different

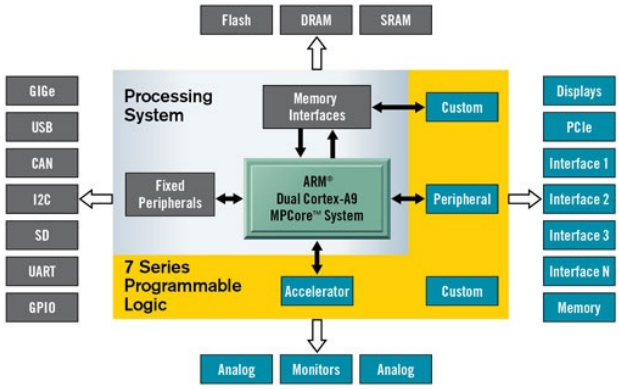

Figure 2. Schematic block diagram of the Zynq SoC [2] with its communication interfaces. The connected peripherals can be managed directly from Python exploiting the PYNQ *base* overlay for PYNQ-Z1.

tools and methodologies to bind functionalities from foreign languages into Python. This means that developers do not have to compromise performance for productivity, as one can always wrap high-performance code written in a lower-level language into Python. This has been proven to be beneficial in the case of PYNQ [7], but it is, in general, an important feature of CPython, already exploited by the community to build sophisticated libraries, such as *NumPy* itself, that expose a simple Python interface but relies on highly optimized code written in another language. Also, PYNQ proposes the concept of *overlays*, or *hardware libraries*, as a mean to utilize the programmable logic. These overlays resemble classical software libraries but expose functionalities of the FPGA. Programmable logic circuits are wrapped as Python modules, that can be imported into the application and allow developers to use hardware functions via a Python API. However, creating an overlay still requires expertise in designing programmable logic circuits. The key aspect is that overlays are conceived to be designed once, but reused multiple times. In this sense, Xilinx's intent is to create an environment where a few experts build overlays to offer a greater user-base the ability to exploit programmable logic while staying at the software level.

C. Embedded Systems Devices

Embedded systems are electronic digital devices specialized in particular functions used as components into larger systems. They monitor and control the system through special hardware devices cheaper and faster than general purpose solutions [8]. The reason why this technology became widespread, not only in large industries but also in everyday life, is related to the IoT (Internet of Things) concept. The IoT paradigm is based on different types of physical devices that are connected together through the network, to allow real-time data transmission and items interaction [9]. What follows is a description of the two platforms we used during our tests and they represent two of the most used devices to develop embedded systems. Moreover, these platforms are used as gateways in already mentioned Fog Computing systems, thanks to their versatility and power. The aim is to decentralize computing infrastructure by extending Cloud Computing and services to the edge of the network [10].

1) **PYNQ-Z1:** The device mainly used in our work is Xilinx PYNQ-Z1[4], that combines Microprocessor and Programmable Logic into a Zynq SoC. Figure 2 shows the Zynq SoC architecture and the available interfaces that can be used through the PYNQ platform. The processor is Dual ARM® Cortex™-A9 MPCore™ with CoreSight™ @ 650MHz with 32 KB Instruction, 32 KB Data per processor L1 Cache, 512 KB unified L2 Cache and 256 KB On-Chip Memory. The available logic is 85K logic cells (13300 logic slices, each with four 6-input LUTs and 8 flip-flops), 630 KB of fast block RAM, four clock management tiles, each with a phase-locked loop (PLL), 220 DSP slices, internal clock speeds exceeding 450MHz. The programmable logic has a logic blocks structure (CLB) surrounded by I/O blocks (IOB) that can be used arbitrarily.
2) **Raspberry Pi 3:** The second device we used is Raspberry Pi 3 model B[5], Broadcom BCM2837 64bit ARMv8 quad-core Cortex A53 processor @ 1.2GHz, 1 Gb RAM, and it belongs to Microprocessor category.

To make a comparison, Raspberry Pi 3 is a valid solution thanks to its reasonable cost, computational power, and energy consumption [11], and its widespread is also thanks to the ease of use. However, PYNQ-Z1 is suitable for the deployment of embedded systems that require high performance to process data. In fact, the programmable logic allows creating custom circuits to accelerate software processes and it can be reprogrammed to be flexible with different use cases (in contrast to Application-Specific Integrated Circuits).

D. Related Work

Kammerl et al. [12] presented two graph-based approaches able to synchronize several audio signals. Features like Spectral Flatness or Zero-crossing Rate are extracted from the audio sources. Then, a pairwise cross-correlation of features is computed to generate the graph. Algorithms like Minimum Spanning Tree or Belief Propagation are then used to compute the final offset of each signal.

Ellis et al. [13] developed a system able to compute music similarity not only adopting feature statistics, but also computing the relative position of those features in tempo-normalized time. It works extracting music features averaged within each beat: this allows to construct the beat-synchronous feature representation. The cross-correlation peak value is the similarity measure between two songs. Shrestha et al. [14] presented two methods able to synchronize videos coming from different sources. This is performed with audio synchronization. One of the methods extracts features from the audio. Then, using a classifier, probabilities of several classes are computed every δt. Each audio class is then compared with the same class of other recordings using cross-correlation. The peak in the correlation coefficient identifies the synchronization point.

[4]http://www.pynq.io/board.html
[5]https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

III. Approach

In this section, we introduce how our solution works and how it can be used to take advantage of the Programmable Logic of Zynq SoCs to accelerate the Audio Signal Alignment application.

A. Transparent Hardware Acceleration

As *NumPy* is the most used library for Python Data Science Application, providing hardware acceleration for it represents a valuable contribution to the PYNQ project. We have therefore built a *NumPy* library designed for the PYNQ platform, to enable transparent hardware acceleration for its Fast Fourier Transform and Cross-Correlation function. Transparency is granted by the fact that using the proposed library boils down to simply changing the name in the import statement, as shown in the following snippet of code. This reduces a lot the effort of final users to take advantage of the Heterogeneous System Architecture (HSA) selected.

```
# Using original NumPy
import numpy as np
...
z = np.correlate(x, y, 'full')
```

```
# Using NumPy with hardware acceleration
import numpynq as np
...
z = np.correlate(x, y, 'full')
```

This is possible thanks to an override-like process of the original *NumPy* functions we used within the *numpynq* module. In this way, if a *NumPy* function is not accelerated in hardware or if the required context is not satisfied, the original software version is called. In the following section, we better explain this concept and its advantages.

B. Runtime Code Scheduling

There are circumstances in which offloading the computation to the FPGA does not bring any benefits, and might actually hurt performance. For this reason, we implement a predictive code scheduling mechanism, with an approach similar to what has been done for GPUs [15]. In particular, for each Fast Fourier Transform or Cross Correlation function call (for which we provide a hardware accelerator), we implement a scheduling policy based on performance history and some input properties or physical constraints. Since our implementation wraps the original *NumPy*, we then automatically delegate to it non-accelerated calls. We mostly consider the input size and the input data type to predict the execution time of the different implementations. We collect performance history data for different inputs and build a model of performance that we then use to discriminate what implementation to choose, given the context.

We depict our code scheduling mechanism in Algorithm 1. In this algorithm, we identify *context()* as the action of extracting contextual information from the specific call, as the input size and the input data type, while *hw_accelerator()* retrieves the available overlay that can be used to accelerate such call if it exists. Finally, *history()* provides an estimation of performance given the current context, relying, as the name suggests, on performance history for the specific hardware accelerator or the software execution of the original NumPy, referenced in the pseudocode as *sw_numpy*. We account also for the reconfiguration overhead for the estimates, checking also whether the FPGA is configured with the considered overlay (and removing the reconfiguration time in such case).

Algorithm 1 Performance History Scheduling

$ctx \leftarrow context(numpy\_call)$
$hw\_impl \leftarrow hw\_accelerator(numpy\_call)$
$chosen\_impl \leftarrow sw\_numpy$
if $history(hw\_impl, ctx) > history(chosen\_impl, ctx)$ **then**
    $chosen\_impl \leftarrow hw\_impl$
return $chosen\_impl$

C. Data Transfers Optimization

To communicate and control DMAs from PYNQ, Xilinx provides the users with a ready-to-use Python class that manages the DMA operations. This interface is written using CFFI[6], that provides the possibility to call C functions, included in properly compiled C libraries, directly from Python. Our preliminary experiments showed that this implementation introduces too much overhead for our purposes, and for this reason, we have decided to re-engineer the DMA communication layer. We opted for rewriting completely the layer using the Python/C API[7] as it offered the best performance, at the cost of a greater implementation effort. Also, we have removed some time-consuming control routines present in the original DMA interface offered by PYNQ when managing the buffers needed for the data transfers. This can be done because of the inclusion of the function-specific control routines in the runtime code scheduling algorithm. Moreover, a lot of control logic is present in the original implementation that is actually superfluous in our case. In fact, with our approach, we hide the DMA interface as everything is handled transparently from the application developer's perspective, while the PYNQ interface is exposed directly to the user.

IV. Proposed Design

In this section we present the hardware/software codesign used to accelerate the Audio Signal Alignment application and the improvements on the Cross-Correlation function and Fast Fourier Transform we obtained following the approach proposed in Section III.

A. Profiling

We used the cProfile Python library[8] to profile the Audio Signal Alignment application with different input audio signals. We decided to accelerate the Cross-Correlation function

[6]https://cffi.readthedocs.io
[7]https://docs.python.org/3.5/c-api/index.html
[8]https://docs.python.org/2/library/profile.html

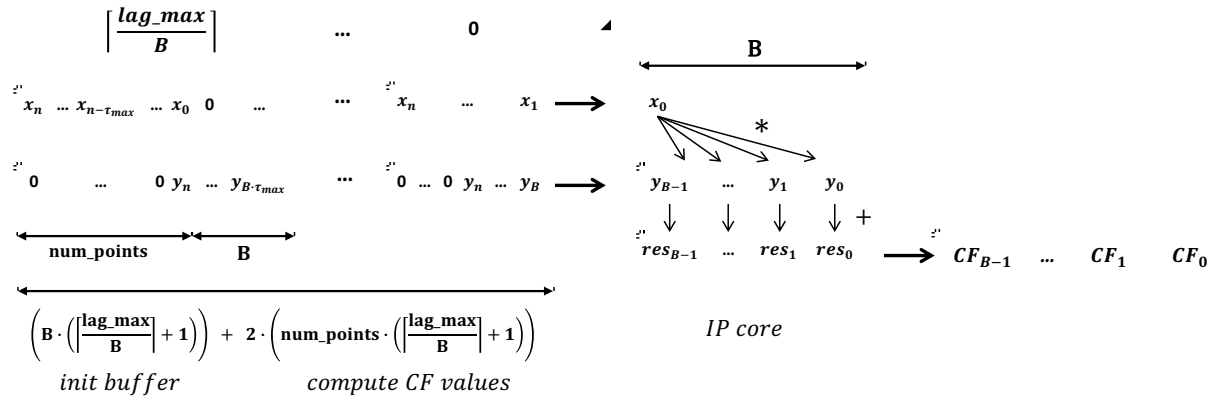

Figure 3. Schema of the FPGA implementation of the Cross-Correlation function. The core has two local buffers to store portions of the signal and compute more values in parallel. In this way the length of the two input streams is reduced by a factor equal to the size of local buffers. At each iteration, the core outputs more than one value.

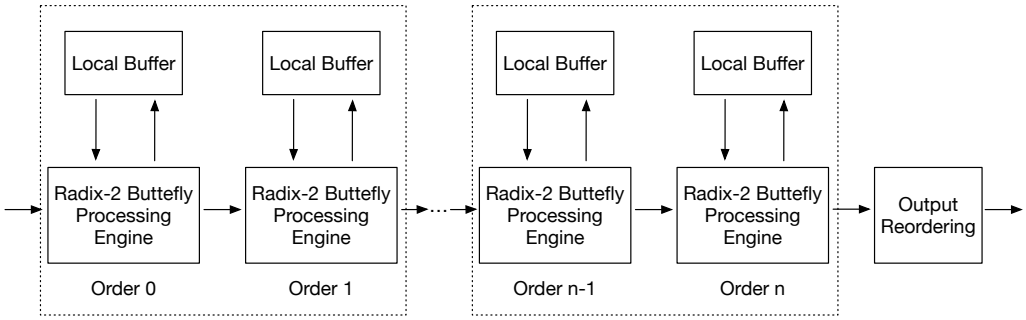

Figure 4. Diagram showing the basic blocks used in the implementation of the hardware accelerator for the Fast Fourier Transform. It implements a streaming and pipelined architecture, consisting of a chain of radix-2 butterfly processing engines. Each engine has its own local memory and is followed by a final output reordering stage at the end of the chain. Unlike cross-correlation, which requires only one core for signals of different lengths, we have implemented several FFT cores to optimize the processing of signals of different lengths. Given an input signal, the right implementation is chosen at runtime by the proposed Runtime Code Scheduling algorithm.

Table I
HARDWARE RESOURCES UTILIZATION

Overlay	LUT	LUTRAM	FF	BRAM	DSP
Correlation	55.87%	1.95%	40.55%	5.36%	38.18%
FFT	19.60%	9.48%	14.68%	76.79%	27.27%

This table reports the post-implementation hardware resources utilization for the two overlays described in this paper. Both of them are created with the Xilinx Vivado Design Suite for the PYNQ-Z1 board. The available resources on this device are: 53200 LUT; 17400 LUTRAM; 106400 FF; 140 BRAM; 220 DSP.

and the Fast Fourier Transform, that occupy respectively 86.73% and 0.33% of the total execution time (the remaining part of the execution time is mainly used by the audio decoding process). This decision was also dictated by the possibility to parallelize the operations that compose these two functions. Taking into consideration that the Cross-Correlation function occupies most of the total execution time, we have chosen to put our maximum effort on its acceleration. We produced two different overlays that can be transparently used at the Python level to speedup the execution time. What follows is the description of the two overlay designs.

B. Cross Correlation function

Given two univariate random signals X, Y, with values $x_1, x_2, ..., x_n$, $y_1, y_2, ..., y_n$ over a time-span $1, ..., n$, and defined a *delay* τ, the Cross Correlation result of the two signals with respect to the delay τ is defined as:

$$CF_\tau = \frac{\sum_{i=1}^{n-\tau}(x_i - \bar{x}_0)(y_{i+\tau} - \bar{y}_\tau)}{\sqrt{\sum_{i=1}^{n-\tau}(x_i - \bar{x}_0)^2}\sqrt{\sum_{i=1}^{n-\tau}(y_{i+\tau} - \bar{y}_\tau)^2}}$$

$$\bar{x}_0 = \frac{1}{n-\tau}\sum_{i=1}^{n-\tau} x_i \qquad \bar{y}_\tau = \frac{1}{n-\tau}\sum_{i=\tau+1}^{n} y_i$$

where $\bar{x}_0$ and $\bar{y}_\tau$ are the sample means of X and Y over interval $n-\tau$.

If the above is computed for all delays $\tau = 0, 1, 2, ...n-1$ then it results in a full Cross Correlation function of twice the length as the original signals X and Y. This function shows

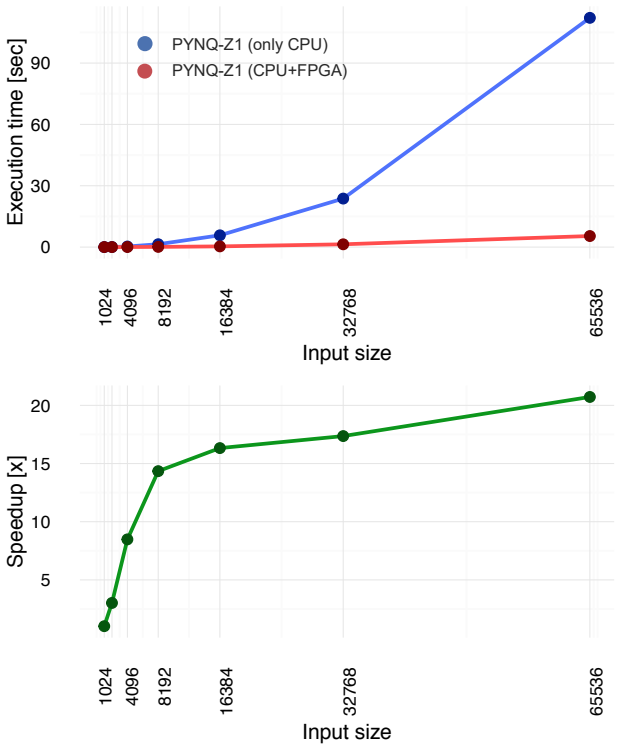

Figure 5. Execution time and speedup of the Cross Correlation function for different signal dimensions. The input size represent the length of the signals. With two signals of size 65536 we have been able to reach a speedup greater than 20x.

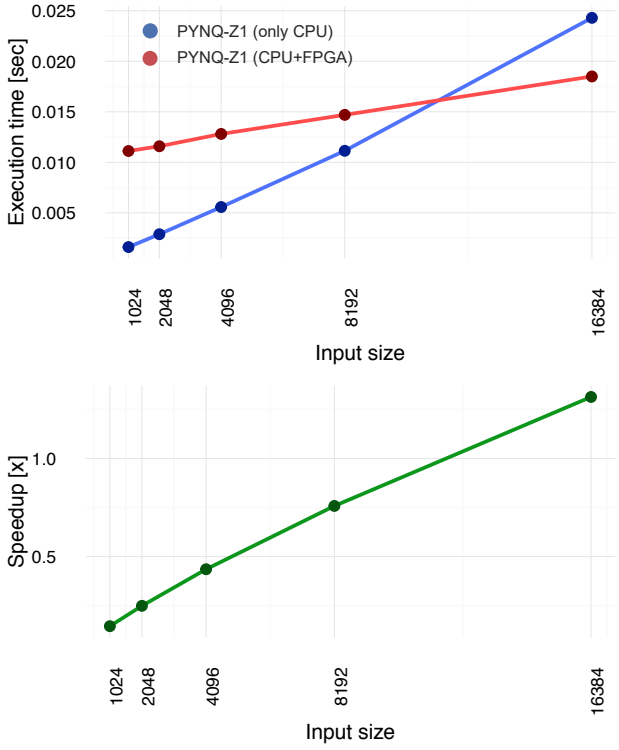

Figure 6. Execution time and speedup of the Fast Fourier Transform for different signal dimensions. The input size represent the length of the input signal. With a signal of size 16384 the hardware accelerated version reaches a speedup of 1.3x.

the degree of similarity of process X with process Y, shifted by a certain delay τ.

To implement the optimized version of this algorithm, we have exploited the fact that it is possible to compute different points of the function independently to parallelize the computation. The schema of our implementation is shown in Figure 3. We have used the Dataflow paradigm to stream the input signals of arbitrary dimensions. In fact, unlike the software implementation where the algorithm accesses each point of the signals from the host DDR using any stride and type of access (with almost no loss in performance due to host cache and pre-caching mechanisms), on the programmable logic random accesses to the DDR on board is a costly operation that can take hundreds of clock cycles. To solve this, it is possible to take advantage of registers (in the form of Look-up Tables) and BRAMs, which, however, are available in limited quantities. So we created two local buffers of size B on BRAMs: the first used to store part of the first input signal, acting as a *shift register*, and the second used to collect partial results. The two streams contain the signals repeated and shifted of a certain delay and are read iteratively. At the end of each iteration, result points are sent from the core to the shared DDR memory through an output stream. In order to keep the two streams synchronized, we pad with zeros one of the two inputs. Moreover, we further improved the design by applying classical pipelining and loop unrolling optimization to guarantee parallel execution and to mask the latency of the operations. The BRAMs containing the local buffers are partitioned to allow parallel accesses during the iterations.

Nevertheless, as many operations are done in parallel, the execution time of the algorithm is proportional to the input streams size: the local buffers reduce the stream size by a factor B, which in turn reduces the complexity of the algorithm by the same factor. Being able to use bigger local buffers should lead to even higher performance improvements. Moreover, our implementation can be easily scaled with respect to the available resources on the board, by changing the local buffer size and its partitioning factor.

Figure 5 shows the results we have obtained with our implementation with respect to the original NumPy Cross Correlation function execution. In particular, we tested the function call both when only the processing system (CPU) of the PYNQ-Z1 is used and when also the programmable logic (FPGA) is exploited. The FPGA implementation achieves a clock frequency of 100MHz and uses 32bits floating points. Thanks to the described design, we have been able to achieve a speedup greater than 20x. The hardware resources utilization is presented in Table I.

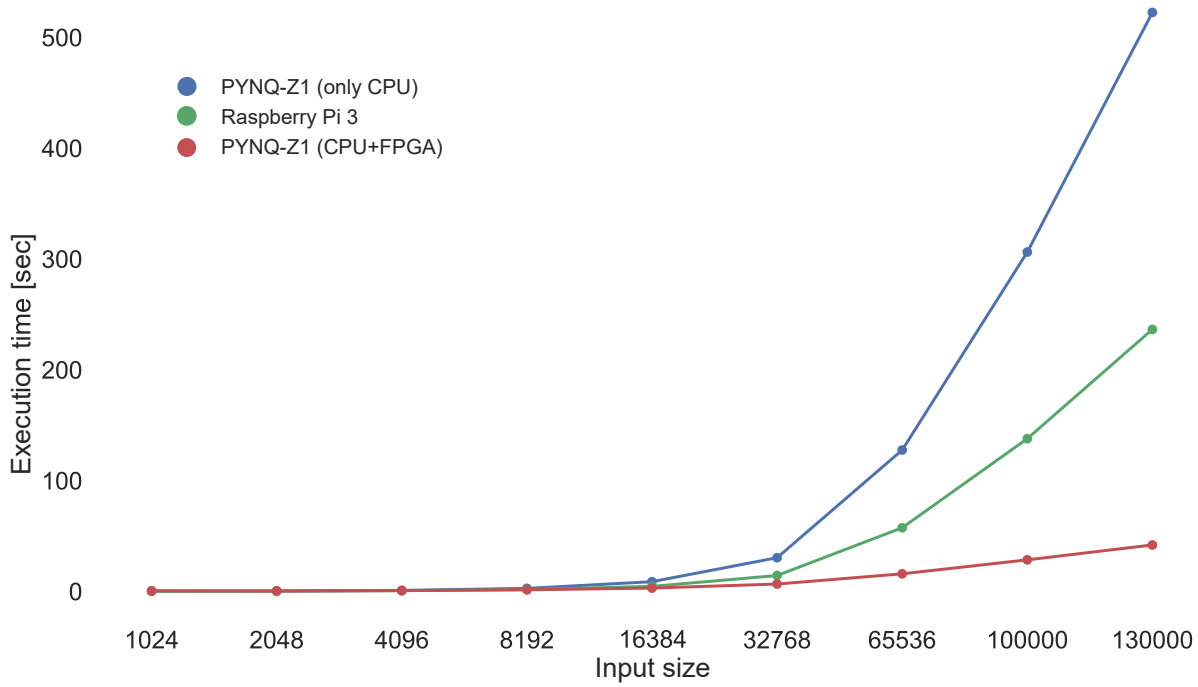

Figure 7. Execution time comparison between Raspberry PI 3 and PYNQ-Z1 with default *NumPy* library, and PYNQ-Z1 with our accelerated implementation that exploits the programmable logic of the Zynq SoC. While considering two audio signals of size 130000, our solution is able to reach a speedup of 12.4x with respect to the PYNQ-Z1 (only CPU) version and a speedup of 5.5x with respect to the Raspberry Pi 3 version.

C. Fast Fourier Transform

The second implementation we present is for the computation of the Fast Fourier Transform. Similarly to the Cross-Correlation accelerator presented, we opted for a streaming computation pattern, to allow continuous data processing.

Figure 4 shows the diagram of the Fast Fourier Transform hardware accelerator, while the hardware resources utilization is presented in Table I. More specifically, we relied on Xilinx's FFT IP core [16], and configured it to implement a streaming and pipelined architecture, consisting of a chain of radix-2 butterfly processing engines. Each engine has its own local memory and is followed by a final output reordering stage at the end of the chain.

Figure 6 shows the comparison between the pure software and the hardware/software execution times. The FPGA implementation achieves a clock frequency of 100MHz and each point of the input signal is represented by a complex number that uses 32bits floating points both for the real and the imaginary component. With a signal of size 16384, the hardware accelerated version reaches a speedup of 1.3x. We note that with a signal of size 8192 the CPU-only execution time is smaller than the hardware implementation. This happens because the time overhead introduced by the data transfer between the DDR and the programmable logic is greater than the time gained by the accelerator in the computing phase. However, thanks to our Runtime Code Scheduling algorithm, the right implementation is chosen at runtime based on the length of the signal.

V. Results

This section describes the evaluation settings we adopted to test our application and reports the results comparison in terms of execution time. Since we want to check if our solution is valid, not only with respect to the PYNQ-Z1 processor but also with respect to a different embedded system device we have tested our implementation on a Raspberry Pi 3 (see Section II-C).

We have run the tests in the following way: we took two audio files of the same conference from AMI Dataset EN2001a [17]. These audio files have been recorded by two different microphones placed in different positions. We then extracted several times random samples: one of this is the shortest one, the other one is longer and includes the same portion of the conference that has been recorded in the first audio. We then added a random zero-mean Gaussian noise to both signals and we repeated these tests multiple times for each different length of the samples. We then computed averages of the overall execution time and of the final error.

The results are presented in Figure 7. As shown in the graph, our solution outperforms both the PYNQ-Z1, when it

runs the pure software implementation of the functions and the Raspberry Pi processor. Specifically, with two audio signals of size 130000, the FPGA accelerated implementation is able to reach a speedup of 12.4x with respect to the PYNQ-Z1 (only CPU) and a speedup of 5.5x with respect to the Raspberry Pi 3. The results also include the reconfiguration time of the FPGA when a new overlay is required. As expected, the Runtime Code Scheduling choose the right implementation to keep the execution time of the PYNQ-Z1 (CPU+FPGA) solution smaller or equal than the PYNQ-Z1 (only CPU) one.

As already said, the Cross-Correlation computation is the most time-consuming aspect of the algorithm, that is why we can clearly notice a much faster execution of the application when FPGA is used. This fact is relevant in the audio field since we usually have a high sample rate frequency. Our results also show that the hardware implementation percentage error is not bigger than the software one. This happens because hardware implementation approximation is quite small and does not change the value in which the peak is located in the Cross-Correlation.

Finally, this experimental setting describes how this algorithm performs quite well in low-noise environments, but this hardware implementation could be used also in more complex and less noise-sensible algorithms, such as the Adaptive Cross-Correlation Method explained in [5]. In these kinds of algorithms, we can expect similar or higher speedup since they require multiple Cross Correlation computations. In case of more than two signals, the speedup will be even bigger.

VI. Discussion and Future Directions

Regarding future work, we are thinking to pack the two IP Cores within the same overlay, to reduce the reconfiguration time overhead. Moreover, we want to take into consideration partial dynamic reconfiguration of the programmable logic to better exploit the FPGA resources. While doing this, an improvement of our Runtime Code Scheduling is required to find an optimal solution for the application execution, following different reconfiguration patterns.

We also will continue to optimize more scientific functions following the proposed approach to offer an increasingly complete solution for data scientists and software developer. We want to compare the power efficiency of the embedded system devices when our solution is running on FPGA with respect to the pure software execution.

Finally, to try our application in a real-time scenario, we are planning to exploit our optimized version of the application to align multiple electrocardiographs measured from an audience to compute the overall quality of attention.

VII. Conclusions

We believe that scientists and pure software developers should be allowed to benefit from hardware acceleration while focusing on what it is most important to them, without the need to invest precious time in learning how to design and deploy hardware accelerators. For this reason, we have proposed in this paper an hardware-accelerated version of the Audio Signal Alignment application, that brings transparent hardware acceleration on Zynq SoCs, when integrated within the PYNQ platform. We described our approach and our function-optimization workflow by showing the implementation processes and the results of two different overlays: the Cross-Correlation function and the Fast Fourier Transform.

To demonstrate the validity of our solution, we have compared the execution times of the different implementations with three different system settings: the PYNQ-Z1 board and the Raspberry Pi 3 device when only the CPU is used, and the PYNQ-Z1 board when both the CPU and the FPGA are exploited. The application that uses our FPGA-optimized version of the application reaches a speedup of 12.4x with respect to the PYNQ-Z1 software execution and a speedup of 5.5x with respect to the Raspberry Pi 3 software execution.

References

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014.

[2] "PYNQ: Python Productivity for Zynq," http://www.pynq.io (accessed: 24th of September 2017).

[3] "IEEE Spectrum: The 2017 Top Programming Languages," https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages (accessed: 5th of October 2017).

[4] I. Stojmenovic and S. Wen, "The fog computing paradigm: Scenarios and security issues," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*. IEEE, 2014, pp. 1–8.

[5] K. J. Coakley and P. Hale, "Alignment of noisy signals," vol. 50, pp. 141 – 149, 03 2001.

[6] T. Giannakopoulos, "pyaudioanalysis: An open-source python library for audio signal analysis," *PloS one*, vol. 10, no. 12, 2015.

[7] A. G. Schmidt, G. Weisz, and M. French, "Evaluating Rapid Application Development with Python for Heterogeneous Processor-based FPGAs," in *Proceedings of the 25th International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '17. IEEE, 2017.

[8] I. of Electrical and E. Engineers, "Ieee standard for information technology-standardized application environment profile-posix realtime application support (aep)," 1999.

[9] M. P. A. Hukeri and M. P. Ghewari, "Review paper on iot based technology," 2017.

[10] N. Constant, D. Borthakur, M. Abtahi, H. Dubey, and K. Mankodiya, "Fog-assisted wiot: A smart fog gateway for end-to-end analytics in wearable internet of things," *arXiv preprint arXiv:1701.08680*, 2017.

[11] W. Anwaar and M. A. Shah, "Energy efficient computing: A comparison of raspberry pi with modern devices," *Energy*, vol. 4, no. 02, 2015.

[12] J. Kammerl, N. Birkbeck, S. Inguva, D. Kelly, A. J. Crawford, H. Denman, A. Kokaram, and C. Pantofaru, "Temporal synchronization of multiple audio signals," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2014, pp. 4603–4607.

[13] D. P. W. Ellis, C. Cotton, and M. Mandel, "Cross-correlation of beat-synchronous representations for music similarity," 03 2008, pp. 57 – 60.

[14] P. Shrstha, M. Barbieri, and H. Weda, "Synchronization of multi-camera video recordings based on audio," in *Proceedings of the 15th ACM International Conference on Multimedia*, ser. MM '07. New York, NY, USA: ACM, 2007, pp. 545–548. [Online]. Available: http://doi.acm.org/10.1145/1291233.1291367

[15] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, "Predictive runtime code scheduling for heterogeneous architectures." *HiPEAC*, vol. 9, 2009.

[16] Z. Jin, L. Jun, and Z. Shuang, "The design and implementation of fft algorithm based on the xilinx fpga ip core," 2012.

[17] "AMI Dataset," http://groups.inf.ed.ac.uk/ami/ (accessed: 27th of April 2018).