Enabling Transparent Hardware Acceleration on Zynq SoC for Scientific Computing

Luca Stornaiuolo
luca.stornaiuolo@polimi.it
Politecnico di Milano
Milan, Italy

Filippo Carloni
filippo.carloni@mail.polimi.it
Politecnico di Milano
Milan, Italy

Riccardo Pressiani
riccardo.pressiani@mail.polimi.it
Politecnico di Milano
Milan, Italy

Giuseppe Natale
giuseppe.natale@polimi.it
Politecnico di Milano
Milan, Italy

Marco Santambrogio
marco.santambrogio@polimi.it
Politecnico di Milano
Milan, Italy

Donatella Sciuto
donatella.sciuto@polimi.it
Politecnico di Milano
Milan, Italy

ABSTRACT

In a quest for making FPGA technology more accessible to the software community, Xilinx recently released PYNQ, a framework for Zynq 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 enable transparent hardware acceleration for scientific computations for Zynq. We do so by providing a custom NumPy library designed for PYNQ, as it is the de-facto scientific library for Python. We then demonstrate the effectiveness of the proposed approach on a biomedical use case involving the extraction of features from the Electroencephalography (EEG).

KEYWORDS

Zynq, PYNQ, Python, NumPy, FPGA

ACM Reference Format:
Luca Stornaiuolo, Filippo Carloni, Riccardo Pressiani, Giuseppe Natale, Marco Santambrogio, and Donatella Sciuto. 2019. Enabling Transparent Hardware Acceleration on Zynq SoC for Scientific Computing. In *Proceedings of EWiLi '19, October 17, 2019, New York, USA,* , (Ed.). ACM, New York, NY, USA, 6 pages.

1 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. However, no matter how much FPGA technology has matured, the usability barrier is still preventing 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, it still substantially requires to go through the same development process.

In an effort to address the usability challenge, Xilinx recently released PYNQ (PYthon productivity for zyNQ)[2]. This project, as the name suggests, is meant to enhance productivity on Zynq SoCs, that integrate a multi-core ARM processor with an FPGA into a single chip, with the help of the popular Python programming language [2]. With Python, 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.

In this work, we start with the observation that a substantial amount of Python applications rely on *NumPy*[3], arguably the most popular Python package, to perform scientific computations, whose computational kernels are usually amenable to hardware acceleration. We leverage this observation and exploit the overlay concept to build a hardware accelerated version of NumPy that is seamlessly integrated into PYNQ. This modified NumPy can be used to accelerate applications by simply changing an import in the end-user application, with everything handled automatically and transparently. We use a biomedical use case involving the extraction of features from the EEG to demonstrate the effectiveness of the proposed approach. This paper represents also an evolution and integration of our past work [3].

The paper continues with Section 2, where we provide an overview of PYNQ and discuss the related work. Section 3 is then devoted to the description of the proposed framework to enable transparent hardware acceleration, and it is followed by Section 4, that discusses the use-case employed for validation. We then present the experimental results in Section 5, before concluding and outlining possible future directions with Section 6.

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

[2]http://www.pynq.io
[3]http://www.numpy.org

EWiLi '19, October 17, 2019, New York, USA,
Copyright held by Owner/Author(s).
2019.

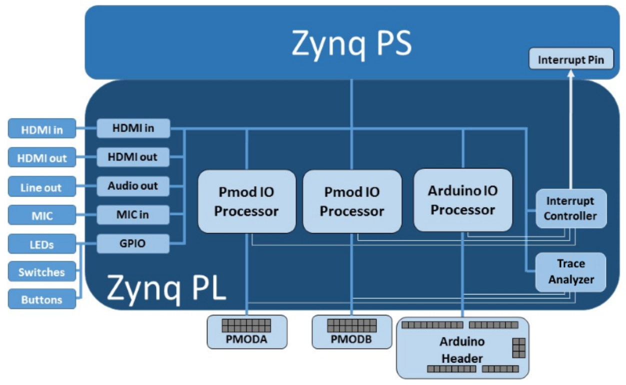


Figure 1: Block design of the official PYNQ-Z1 *base overlay* from Xilinx. The design includes hardware IP to control GPIO devices (LEDs, Switches, Buttons), Video, Audio, and other custom interfaces on the target board, and connects these IP blocks to the Zynq PS.

2 PRELIMINARIES

2.1 The PYNQ Framework

PYNQ has been released in 2016 by Xilinx, it targets Zynq SoCs, and its objective is to allow developers to write applications that exploit the programmable logic without having to use, or know at all, 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 [2], 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 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 [4], 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 means 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 HW 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 a funnel of developers in a way that mimics Linux development. The development of the Linux kernel is done by a handful of highly-skilled developers that enable, with their contributions, the majority of software developers to build their applications with a higher level of abstraction. The rationale behind overlays is exactly the same: few experts that build overlays to offer a large user-base the ability to exploit programmable logic while staying at the software level.

The PYNQ environment includes by default a single overlay, called *base overlay*, that is designed to target a specific class of users: the community of "makers" familiar with Raspberry Pi or Arduino. The block design is shown in Figure 1. This overlay comes with support for two 12-pin PMOD connectors and an Arduino-compatible interface, that can be used to connect Arduino shields, for handling generic I/O and communicating with external devices. Audio and HDMI I/O are also supported, but the overlay provides only the controllers, no HW modules have been designed to process audio or video signals using the programmable logic. This overlay instantiates 3 MicroBlaze soft-cores, one to drive the I/O for the Arduino-compatible connector, and one for each PMOD. The Python *pynq* package also offers APIs to manage the loading of overlays, with mechanisms to hold information about the current overlay, used to implement safety measures and perform run-time management, and to directly access the FPGA through Memory-Mapped I/O (MMIO) and Direct Memory Access (DMA) transactions.

2.2 Related Work

As stated before, PYNQ has been made publicly available only recently. Nonetheless, a research effort to exploit and extend such framework has already started. For instance, in [4], the impact of PYNQ usage on Zynq SoCs is evaluated, finding potential benefits in development time and performance. In [5], the framework has been extended with support for dynamic partial reconfiguration. In [6], PYNQ is integrated with Spark and evaluated on a machine learning application.

The idea of exploiting NumPy to enable transparent hardware acceleration is also not new. Indeed, although no previous work, to the best of our knowledge, targets FPGAs, there are multiple attempts in the literature to use NumPy to enable seamless acceleration on GPUs [7, 8].

It is also interesting to notice that there are several projects that rely on Python to design hardware [9–11], although they utilize the Python syntax for low-level hardware description, in a way that is functionally equivalent to using classical Hardware Description Languages (HDLs). However, this approach is transversal to the approach proposed in this paper, as we rely on overlays, *i.e.* where the design of the hardware accelerator is done previously by an expert and is not part of the application development flow. In this sense, one might even imagine performing the hardware design of the overlays integrated into our modified NumPy library using one of these projects, although this is outside the scope of this work.

3 TRANSPARENT HARDWARE ACCELERATION: NUMPY FOR PYNQ

As NumPy is the de-facto scientific library for Python, providing hardware acceleration for it could be a valuable contribution to the PYNQ project. We have therefore built a NumPy library specifically designed for PYNQ, to enable transparent hardware acceleration for a number of its core functions. Transparency is granted by the fact that using the proposed library boils down to simply changing the name in the import statement.

In the rest of this section, we provide some details of the most important aspects that characterize this work.

3.1 Iterative Runtime Code Scheduling

There are circumstances in which offloading the computation to the FPGA does not bring any benefits, and might hurt performance. For this reason, we implement a predictive code scheduling mechanism, with an approach similar to what has been done for GPUs [12]. In particular, for each NumPy call for which we provide one or more hardware accelerators, 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 unaccelerated calls to it. 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 chose 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_accelerators()* retrieves all available overlays that can be used to accelerate such a call. Finally, *history()* provides an estimate 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

```
$ctx \leftarrow context(numpy\_call)$
$hw\_list \leftarrow hw\_accelerators(numpy\_call)$
$chosen\_impl \leftarrow sw\_numpy$
**for all** $hw\_impl \in hw\_list$ **do**
    **if** $history(hw\_impl, ctx) > history(chosen\_impl, ctx)$ **then**
        $chosen\_impl \leftarrow hw\_impl$
**return** $chosen\_impl$
```

3.2 Matrix Dot Product Example

As part of this work, we have developed a number of overlays to supply hardware accelerators for the most common *NumPy* calls. To showcase the hardware acceleration flow and the steps needed to include the optimized version of the function within the PYNQ platform we firstly present the improvements we made to the *NumPy* matrix dot product. In particular, we developed two overlays for two different contexts, with the purpose of exploiting the reconfigurable logic in the best possible way every time. The first overlay supports calls for arbitrary input size and exploit hardware pipeline with a streaming pattern to achieve a speedup with respect to the software execution, the second one performs the computation for matrices where the size fits in the BRAMs, so that we can load the entire input signals on the FPGA and compute multiple operations in parallel. Then, we leverage the Runtime Code Scheduling to choose the right implementation to call, when an application is executed. In this example, we decided to accelerate the matrix dot product for *integer* numbers.

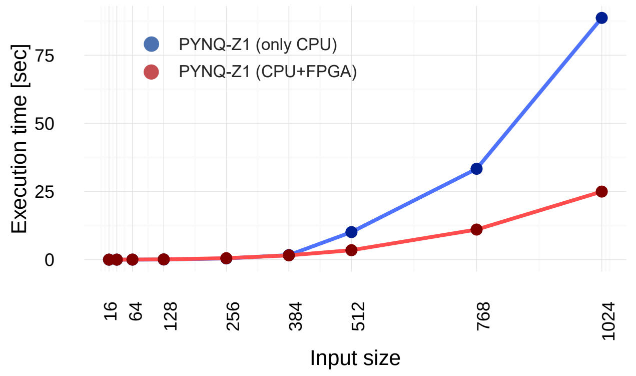


Figure 2: Execution time of the Pipelined Integer Matrix Dot Product for different matrix dimensions. The input size represents the dimension *N* of two square matrices, each of size *(N x N)*. The break-even point is approximately around an input size of 384. With two matrices of size *(1024 x 1024)* the hardware accelerated version reaches a speedup of 3.5x.

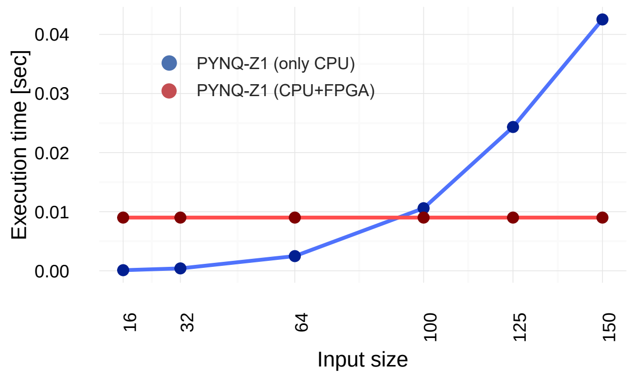


Figure 3: Execution time of the Parallel Integer Matrix Dot Product for matrix of dimensions up to 150. The input size represents the dimension *N* of two square matrices, each of size *(N x N)*. The break-even point is approximately around an input size of 100. With two matrices of size *(150 x 150)* the hardware accelerated version reaches a speedup of 6.1x.

3.2.1 Pipelined Integer Matrix Dot Product. For the integer matrix dot product of arbitrarily large input size, we have created two input streams to send the rows of the first matrix and the columns of the second one repetitively. The multiply-and-accumulate operations are computed using hardware pipelining and the result is stored in on-chip buffers. At each clock cycle, one point from each stream is read and, after a number of points equal to the first matrix's row size (by assumption equal to the second matrix's column size), the result is sent through the output stream. The behavior of the

Pipelined Integer Matrix Dot Product for two matrices of arbitrary input size, A of dimension $(n \times m)$ and B of dimension $(m \times p)$, is also explained by the following pseudo-code, inspired from the Vivado HLS version we produced for the implementation.

```
void pipelined_dot(stream<int> &a, stream<int> &b,
                   stream<int> &c) {
    accum = 0;

    [...] // read n, p and m parameters from the input streams

    // iterate until all input points are read
    for (int i = 0; i < n*p*m; i++) {
        #pragma HLS PIPELINE II=1
        accum = accum + a.read() * b.read();

        if (i % m == m - 1) {
            c.write(accum);
            accum = 0;
        }
    }
}
```

3.2.2 Parallel Integer Matrix Dot Product. If we consider small matrices that can be stored by only using registers and BRAMs of the FPGA, it is possible to significantly reduce the number of input points passed to the IP-Core, by avoiding duplication of the rows and columns, and to better exploit the hardware levels of parallelism to compute results. The following snippet of pseudo-code describes how to parallelize the computation by partitioning the local BRAMs of the board. We have fixed the matrices dimension to (150×150) with a partitioning factor of 50. The choice of the local buffers dimension and of the partitioning factor depends on the number of hardware resources available. One peculiarity of this design is that the dot product for fixed size matrices can be used also with matrices of smaller dimensions by padding them with zeros. The implementation is inspired by [13]. The unroll optimization of the inner-most loop allows performing multiple operations at the same time exploiting the hardware resources.

```
#define DIM 150

void parallelized_dot(stream<int> &s_in, stream<int> &s_out) {
    int a[DIM][DIM], b[DIM][DIM], c[DIM][DIM];
    int const FACTOR = 50;
    #pragma HLS PARTITION variable=a factor=FACTOR
    #pragma HLS PARTITION variable=b factor=FACTOR

    [...] // stream in input matrices

    // matrix multiplication of a A*B matrix
    // in parallel
    L1:for (int ia = 0; ia < DIM; ++ia)
        L2:for (int ib = 0; ib < DIM; ++ib) {
            #pragma HLS PIPELINE II=1
            int sum = 0;
            L3:for (int id = 0; id < DIM; ++id){
                #pragma HLS UNROLL
                sum += a[ia][id] * b[id][ib];
            }
            c[ia][ib] = sum;
        }

    [...] // stream out result matrix
}
```

3.2.3 Speedup and Runtime Code Scheduling. Figure 2 and Figure 3 show respectively the execution time and the speedup for the two proposed Integer Matrix Dot Product overlays implemented for PYNQ-Z1 with a clock frequency of 100MHz. The two break-even points - *i.e.* the number of input points for which the execution with the FPGA performs better than the pure software implementation - are respectively 384 for the Pipelined implementation and 100 for the Parallel one. However, the Parallel implementation can be used with dimensions up to 150, due to the hardware design. In this context, based on the given information, our Code Scheduling evaluates the input data type and the matrices dimensions at run-time and executes the first implementation with integer matrices of dimensions greater than 384 and the second one with integer matrices of dimensions between 100 and 150. Moreover, if the device is not configured with the right overlay, the Runtime Code Scheduling adds the PYNQ-Z1 reconfiguration time of 0.45 seconds to the estimated execution time and recomputes the break-even point. If the input does not match the constraints for hardware acceleration, the original *NumPy* function call is executed on the processor.

3.3 Overlays

While the Matrix Dot Product example shown in the previous paragraph has been inserted only for explanatory purposes, we here provide an overview of the overlays employed in the biomedical use case described in Section 4.

3.3.1 Correlation. As far as correlation is concerned, we have exploited the fact that it is possible to calculate different points of the function independently to parallelize the computation. We used the BRAM to create two local buffers, 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 first input stream contains the first signal repeated a number of times equal to the size of the output function divided by the number of parallel operations that can be performed. The second input stream contains the repetition of the second signal shifted by different lags. At the end of reading a number of points equal to the input signals size, result points are sent from the core to the shared memory through an output stream. To keep the two streams synchronized, we pad with zeros the shifted signals. Moreover, to guarantee parallel execution and to mask the latency of the operations, we further improved the design applying classical pipelining and loop unrolling optimizations.

3.3.2 Matrix Dot Product. For the matrix dot product of arbitrary large input size, we employed the Pipelined Integer Matrix Dot Product described in Section 3.2.1.

3.3.3 Standard Deviation. The standard deviation filter is implemented as two consecutive discrete blocks. The first block computes the variance of the input signal, and is a pipelined, windowed implementation. The second block computes the square root of the variance to output the standard deviation. We implemented the second block relying on Vivado HLS and the already optimized implementation of the square root provided by Xilinx with their *hls_math.h* library.

3.3.4 Fast Fourier Transform. The last implementation we present is for the computation of the Fast Fourier Transform (FFT). Similarly to the other hardware accelerators presented, we opted for a streaming computation pattern, to allow for continuous data processing. More specifically, we relied on Xilinx's FFT IP core and configured it to implement a streaming and pipelined architecture, consisting of a chain of radix-2 butterfly processing engines, each engine with its own local memory, followed by a final output reordering stage at the end of the chain.

4 A BIOMEDICAL USE CASE

We validated the work proposed in this paper by proposing an accelerated version of a bio-medical application for signal processing of EEGs. Nowadays, a multitude of Brain Computer Interface (BCI) is being developed to cope with serious diseases (such as Amyotrophic Lateral Sclerosis) or to control artificial prostheses, bypassing the muscular activity [14]. These technologies rely on the EEG as a non-invasive interface thanks to its easy usability, portability and relatively low-cost (especially if compared to really expensive systems such as Magnetoencephalography (MEG) and functional magnetic resonance imaging (fMRI)). However, the analysis of EEG ensemble requires a non-negligible computational power, due to the families of algorithms involved, and the intrinsic characteristics of the EEG signals, consequently limiting both the development of novel BCI and its usage outside of laboratories, in the form of embedded EEG-feature processors. Some of these characteristics are: high number of channels, from 16 up to 256 in high-density arrays; high sampling frequency (250 - 1000 Hz); low amplitude and signal to noise ratio, with respect to other biological signals (e.g. eyes' muscles or heart rate activity). Given that developers are already accustomed to the use of Python libraries for EEG analysis [15], the availability of transparent FPGA-accelerated libraries could speed-up the research process, and at the same time, the underlying hardware libraries could be reused to implement the algorithm on embedded or portable FPGA systems. In this use case, 4 EEG features are calculated, as they are used in the estimation of brain functional connectivity, the level of coordinated activation that appears in neural assemblies or brain structures during a certain task. These features are namely: the *auto-correlation* and the *cross-correlation* of all channels and their *power spectra*, and the *cross-correlation coefficient*. Let us define x_m and y_m as the EEG measurements, for m = 1,2,..,M obtained from the set of available sensors. The discrete correlation of two signals $x_m(n)$ and $y_m(n)$ at lag τ and sample n is:

$$R_{xy}(\tau) = \sum_{n \in Z} x_m(n) \overline{y}_m(n-\tau) \tag{1}$$

with $x_m = y_m$ in the case of auto-correlation. Following, the cross-spectral density is calculated as:

$$P_{xy}(f) = \sum_{l=-\infty}^{\infty} E\{x_m(n)\overline{y}_m(n-\tau)\} e^{-j2\pi\tau f} \tag{2}$$

where f is the frequency and $E\{\cdot\}$ is the expected value. Finally, the cross-correlation coefficient is defined as:

$$\rho_{xy}(\tau) = \frac{1}{\sigma_x \sigma_y} \mathrm{E}\{(x_m(n) - \mu_x)\left(\overline{y}_m(n+\tau) - \mu_y\right)\} = \frac{1}{\sigma_x \sigma_y} \gamma_{xy}(\tau) \tag{3}$$

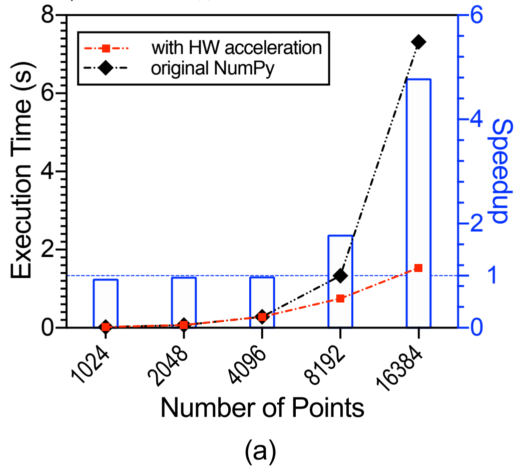
with $\tau = 0$ where γ_{xy} is the *cross-covariance* between the two signals, and $\mu_x, \mu_y, \sigma_x, \sigma_y$ represent the respective mean and standard deviation of the two processes.

These features are useful to understand the degree of connectivity in the brain, in terms of hierarchical structures (*auto-correlation*), general interactions between different areas (*cross-correlation coefficients*) and the coordinated activation at specific frequencies(e.g. the *Beta-waves* related to motor control).

In particular, we chose these features to outline the re-utilization of some calculations, and the possibility guaranteed by the FPGA to pipeline parts of the computation (e.g. the power spectral density after cross-correlation), or the parallel execution on multiple channels through core replication of the FPGA overlay. The capability to switch between different implementations given the input size of the processes is particularly interesting in the case of EEG data, both because the sampling frequencies could span from 250Hz to 1 kHz, with different constraints in terms of computational power, and because BCI signals commonly involve the slicing of the signal in time epochs, ranging between 1-2 seconds (motor tasks, Event Related Potentials (ERP)) to dozens of seconds (resting-state connectivity [16]). As stated before, the entire approach remains completely transparent in the application, optimizing performance without requiring any effort by the end-user in adapting the algorithm.

5 RESULTS

We conducted our experimental evaluation on the PYNQ-Z1 board, the first released board supported by PYNQ. This board comes with a Zynq XC7Z020 SoC, integrating a 650MHz dual-core ARM Cortex-A9 processor and an Artix-7 family FPGA. The use case presented in Section 4 is written in Python and relying on NumPy to perform the EEG features extraction. All it required to accelerate this application with our approach, was changing the import statement for NumPy, with therefore a minimal implementation effort and virtually no impact on the source code of the application. To perform the tests we used real EEG data corresponding to the IVa dataset from the BCI Competition III [17]. The dataset is composed of records of five healthy subjects who were performing motor imagery tasks (the act of mental simulation of a motor action) while sitting on a chair. The brain activity was recorded using an EEG system with 118 sensors located on the scalp according to the 10-20 EEG coordinate system. The data were originally acquired at 1 KHz. We tested the Python application using different-sized input signals, comparing the execution time of the original NumPy within the PYNQ framework against our library. The results are shown in Figure 4a, where we report the two execution times (lines) and the speedup obtained when using our solution (bars). The number of points refers to the amount of sample data of the input signals processed by the application. As shown in the graph, our solution does not bring speedups for 1024, 2048 and 4096 points, where we instead experience a slight increase in execution time due to the overhead of our scheduling mechanism (approximately 5%). In these three instances, the system predicts that using HW acceleration is not beneficial, and delegates the calls to the original NumPy. For 8192 and 16384 points, we instead yield a speedup of respectively 1.77x and 4.76x. As we inspected the results further, we discovered that the speedup was caused by the



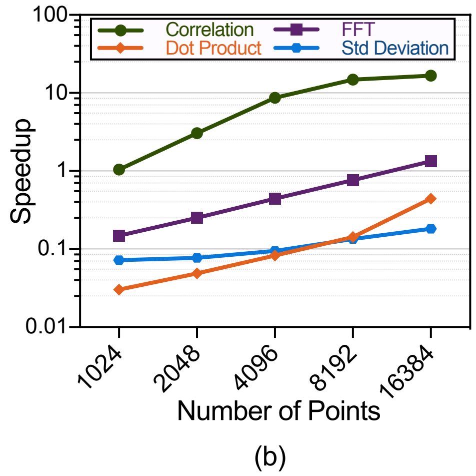


Figure 4: On the top (a), we show the experimental results for the entire biomedical application, where we compare our solution against the original NumPy. On the bottom (b), we report the speedups for the different NumPy calls used in the application.

hardware acceleration of the correlation, the other NumPy calls were indeed not accelerated. The reason for this is shown in Figure 4b, where we plot the speedups over the original NumPy of the different calls with respect to the same input intervals, in log scale. The graph clearly shows that for this number of points we yield a speedup only for the correlation and the FFT. However, as the scheduling system accounts also for the constant reconfiguration time when predicting the performance, during the actual execution the FFT has not been accelerated, since adding the reconfiguration time would have indeed caused a slowdown.

Notice that although using an increasingly high input signal would have eventually yielded a speedup for all the calls, we restricted our tests for signals whose size is meaningful for the real application. It is also worth noting that we evaluated how our DMA layer performs, and we experienced an average 2x speedup with respect to the one offered by PYNQ.

6 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 NumPy that brings transparent hardware acceleration on Zynq SoCs if integrated with the PYNQ framework. To demonstrate the validity of our solution, we evaluated it on a biomedical use case involving the extraction of features from the EEG, performing a comparison with the original NumPy. The results show that our system is able to automatically achieve speedups when hardware acceleration is beneficial, at the cost of a slight decrease in performance when it is not the case, due to the overhead introduced by our scheduling mechanism. Additional PYNQ-based accelerators for *NumPy* library functions will be released in future work to create a complete solution to transparently leverage FPGAs for scientific computing.

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] "IEEE Spectrum: The 2017 Top Programming Languages." https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages (accessed: 5th of October 2017).
[3] L. Stornaiuolo, M. Perini, M. D. Santambrogio, and D. Sciuto, "Fpga-based embedded system implementation of audio signal alignment," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 132–139, IEEE, 2019.
[4] 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*, FCCM '17, IEEE, 2017.
[5] B. Janßen, P. Zimprich, and M. Hübner, "A dynamic partial reconfigurable overlay concept for PYNQ," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, IEEE, 2017.
[6] E. Koromilas, I. Stamelos, C. Kachris, and D. Soudris, "Spark acceleration on FPGAs: A use case on machine learning in Pynq," in *Modern Circuits and Systems Technologies (MOCAST), 2017 6th International Conference on*, IEEE, 2017.
[7] T. Blum, M. R. Kristensen, and B. Vinter, "Transparent GPU execution of NumPy applications," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, IEEE, 2014.
[8] M. R. Kristensen, S. A. Lund, T. Blum, K. Skovhede, and B. Vinter, "Bohrium: unmodified NumPy code on CPU, GPU, and cluster," *Python for High Performance and Scientific Computing (PyHPC '13)*, 2013.
[9] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, IEEE Computer Society, 2014.
[10] P. Haglund, O. Mencer, W. Luk, and B. Tai, "PyHDL: Hardware Scripting with Python," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2003.
[11] E. Logaras, O. G. Hazapis, and E. S. Manolakos, "Python to Accelerate Embedded SoC Design: A Case Study for Systems Biology," *ACM Trans. Embed. Comput. Syst.*, vol. 13, Mar. 2014.
[12] 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.
[13] D. Bagni, A. Di Fresco, J. Noguera, and F. Vallina, "A zynq accelerator for floating point matrix multiplication designed with vivado hls," *Application note, January*, 2016.
[14] I. Choi, I. Rhiu, Y. Lee, M. H. Yun, and C. S. Nam, "A systematic review of hybrid brain-computer interfaces: Taxonomy and usability perspectives," *PloS one*, vol. 12, no. 4, 2017.
[15] A. Gramfort, M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, L. Parkkonen, and M. S. Hämäläinen, "Mne software for processing meg and eeg data," *NeuroImage*, vol. 86, no. Supplement C, 2014.
[16] E. Olejarczyk, L. Marzetti, V. Pizzella, and F. Zappasodi, "Comparison of connectivity analyses for resting state eeg data," *Journal of Neural Engineering*, vol. 14, no. 3, 2017.
[17] G. Dornhege, B. Blankertz, G. Curio, and K. R. Muller, "Boosting bit rates in noninvasive eeg single-trial classifications by feature combination and multiclass paradigms," *IEEE Transactions on Biomedical Engineering*, vol. 51, June 2004.