2017 IEEE International Parallel and Distributed Processing Symposium Workshops

Exploiting FPGAs from Higher Level Languages A signal analysis case study

L. Stornaiuolo, A. Parravicini, G. Durelli, M. D. Santambrogio
Politecnico di Milano, DEIB, Italy
{luca.stornaiuolo, alberto.parravicini}@mail.polimi.it
{gianlucacarlo.durelli, marco.santambrogio}@polimi.it

***Abstract*—Field Programmable Gate Arrays (FPGAs) are usually perceived as difficult to exploit due to the High Level of expertise required to program them. In the last years, the major FPGAs vendors have produced different High Level Synthesis (HLS) tools to help programmers during the flow of acceleration of their algorithms through the hardware architecture. However, these tools often use languages considered low level from the point of view of data scientists and are still much too difficult to use for software developers. This complexity limits their usage in a number of fields, from data science to signal processing, where the computational power offered by FPGAs could be highly beneficial. One way to overcome this problem is to realize libraries of widely used algorithms that transparently offload the computation to the FPGAs device from modern High Level Languages. Our work presents an interface between R, a language commonly used by statisticians and data scientists, and an FPGA connected via PCI-Express (PCIe). We use the Reusable Integration Framework for FPGA Accelerators (RIFFA) to send and receive data from PCIe connection. To showcase the use of the described interface and the improvements given by making use of FPGAs in signal analysis applications we used Xilinx Vivado Design Suite to implement an accelerated and optimized version of the Autocorrelation Function (ACF) present in the default libraries used by R.**

I. INTRODUCTION

The interest for data-mining, machine learning and signal analysis has been growing steadily with the need of tools that can process large data, at higher and higher speeds. The natural consequence has been a shift from traditional architectures, to High-Performance Computing (HPC) [1] systems with an increasingly high amount of parallelism. However, exploiting these architectures to accelerate applications is a long and hard process that requires time to learn appropriate techniques to get advantages from the hardware and to prototype and test different implementations. Because of these reasons many data scientists still use old tools and libraries of sequential algorithms. In this paper, we consider as Higher Level Languages those programming languages with strong abstractions in the context of data science and signal analysis. In these languages, the users can use the algorithms they need simply by importing the needed libraries, without a deep knowledge of software design. Examples of these languages are Python, MATLAB and R.

Modern tools aim at bridging the gap between simplicity and performance. They allow users to remain in the comfort of Higher Level Languages while taking advantage of the parallelism of hardware architectures. Nowadays, Graphic Processing Units (GPUs) are the most popular choice for improving the performance of data analysis. This is due to a combination of multiple factors, such as their availability and the strong investments made by NVIDIA to promote the parallel computing platform CUDA [2]. The result is a robust support to the most widely used libraries in data analysis and computational sciences, and a strong integration with the most used Higher Level Languages. The interest of data scientists towards Field Programmable Gate Arrays (FPGAs) has recently spiked, due to the computational power, the performance per watt and the reconfigurable flexibility [3] offered by these architectures. These properties allow also the usage of the FPGAs to realize embedded systems without involving any other device [4]. However, they are still seen by many as obscure and hard to program, which results in a low amount of applications being ported to these platforms.

Even though Computer-Aided Design (CAD) tools to target FPGAs are improving, there is still a gap between the solutions provided by these CAD tools and the needs of data scientists. Over the last years, in fact, High Level Synthesis (HLS) tools allowed to rapidly develop IP cores and accelerators starting from C code or other languages instead of writing pure Hardware Description Language (HDL) code; however the result of this process still need to be integrated at system level and a set of libraries needs to be written to be ready to use by data scientists.

The topic of this work is to illustrate a first prototype, using a case study from signal analysis, of an architecture and libraries to integrate a FPGA in the context of a language commonly used by data scientist such as R. In particular, we will illustrate:

- how the Autocorrelation Function (ACF) algorithm has been implemented for FPGA using Vivado HLS and how it has been optimized to improve its performance;
- how the IP core has been integrated at System On Chip (SOC) level to allow communication with the host system running R;
- how the IP core can be then exploited within the R flow;

Our results will show that such implementation improves over the implementation of the ACF function available in the R library.

The paper is organized as follows. Section II describes some of the works related to the one of this paper and clarifies the context of the proposed solution. Section III presents the case study and its analysis. Section IV illustrates how the analyzed algorithm have been implemented on the target FPGA. Section V describes how we performed system level integration, realizing an HW architecture able to communicate over PCI express with the host system; then Section VI illustrates how the realized architecture can be exploited from Higher

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.68

Level Languages. The results are presented in Section VII. Finally, Section VIII draws the conclusions and present future directions of work.

II. Related works

The need of using alternative solutions to standard Central Processing Unit (CPU) and multicore architectures in the field of data science and signal processing is caused by the availability of unprecedented amount of data to process [5] that allows for an efficient exploitation of highly parallel architectures such as GPUs and FPGAs. Both these architectures can be configured to perform Single Instruction Multiple Data computation [6] to accelerate the processing of a large amount of data; however FPGAs can also exploit instruction level parallelism if the designer realizes computational pipelines and such approach can also allow to hide data transfer latency.

GPUs started to be used for scientific computation since NVIDIA released the CUDA framework [2]. Thanks to this framework, general purpose computation can be executed on GPU, while they were used only for graphic processing before. Higher Level Languages rapidly began to exploit this capability by realizing libraries to efficiently exploiting GPU almost without the user being aware of it. For example, a wide range of MATLAB functions can be transparently executed on the GPU; the user has only to create the input data calling a specific function (called $gpuArray()$) that allocates data inside of the GPU DDR memory instead of the CPU one. In other languages as Python libraries such as PyCUDA have been released for the same purpose [7].

For FPGAs instead such level of abstraction is still not available and FPGAs are not exploited in scientific computing except for rare situations where ad hoc solutions have been realized [8]. Often the FPGA-based accelerator is interfaced with the CPU through a host function, written in C/C++, that sends data to the FPGA device and receives back results [9, 10]. This is also the way to interface an FPGA accelerator that exploits the OpenCL standard [11, 12]. One of the aims of this paper is to communicate with the host function transparently from the user application. In general, every Higher Level Language has the possibility to connect to external libraries and languages by providing specific libraries for converting its internal objects to the ones of the target library or languages. This is the case for MATLAB with the MEX files [13, 14], for Python with Boost-Python [15], and R with Rcpp [16]. These solutions, or similar ones, are the starting point when a Higher Level Language has to be extended to support external libraries and components as we did in this paper. Moreover, it is possible to connect different Higher Level Languages to the same host function and exploit the build once re-use many times paradigm simply by creating a specific interface for each language to be connected.

However, the benefits of FPGAs for scientific algorithms have been demonstrated multiple times by works implementing accelerators for different problems. One example is the data mining field, where, over the last years, multiple works have proposed FPGA solutions to the implementation of clustering algorithms. As an example, an implementation of a K-Means algorithm has been proposed in [17], while [18] presents a solution for DBSCAN. The implementation of the Autocorrelation Function (ACF) of the R library proposed in this paper aims to illustrate how it is possible to take advantages from the exploitation of FPGA by illustrating some techniques commonly used for this purpose. Some related works about Crosscorrelation Function are proposed in [19, 20], but they use different variants of the algorithm to achieve different results.

In the context of simplifying the development of IP cores and the runtime for FPGAs in general, a solution that is gaining traction over the last years is the possibility to target FPGA by starting from the OpenCL language [21]. This solution is the one adopted by Xilinx with SDAccel tool [22, 23]. Our solution is an initial prototype of a hardware system that resembles the one supported by SDAccel framework, with the advantage that we are not restricted by the need for another tool and design flow, as it is the one in SDAccel, and we can rely on the standard flow used in Xilinx Vivado tool suite. Furthermore, SDAccel support is limited to a small range of devices, while the solution proposed here potentially supports all the devices for which RIFFA is available, which means that the classic development board available in the research community will be easily supported.

III. Case Study: Autocorrelation Function

We decided to use as case study for this work an algorithm from the signal processing field: the ACF algorithm. This algorithm is not only important per se, but it is also a building block of other algorithms such as the Principal Component Analysis or in general algorithms that perform dimensionality reduction on datasets by selecting only the most relevant/descriptive features. This section presents the definition of the algorithm, as well as the description of its implementation in R, the analysis of its complexity obtained with an initial profiling phase and some consideration of why this algorithm is suitable for being accelerated on FPGA

A. Definition

Given two univariate random processes 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 *lag* τ, the empirical (or sample) Correlation Function (CF) $\hat{\rho}_{X,Y}(\tau)$ is defined as:

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

with $\bar{x}_0 = \frac{1}{n-\tau}\sum_{i=1}^{n-\tau} x_i$ and $\bar{y}_\tau = \frac{1}{n-\tau}\sum_{i=\tau+1}^{n} y_i$, the sample means of X and Y over interval $n - \tau$. By changing the value of τ, we model the empirical CF of the processes X and Y, which shows the correlation between the processes at various times. The CF shows the degree of similarity of process X with process Y, shifted by a value τ. If process X is equal to process Y, we get the empirical Autocorrelation Function (ACF) of Y, that represents the internal similarities of the process with itself. Under the hypothesis of equispaced observations, one can replace X with Y in the above formula to compute the ACF values of Y and have information about the randomness of the process. This helps to identify an appropriate time series model (if several *lag* values are analyzed) [24].

B. R Implementation

The default ACF implementation, present in R libraries as part of package *stats*, processes the input signal into the main routine written in R and then calls a subroutine written in C that computes and returns the final ACF values. The main parameters are a process Y with values $y_1, y_2, ..., y_n$ over a time-span $1, ..., n$ (defined as a univariate or multivariate time series or a numeric vector or a matrix) and the desired maximum lag (lag_max).

The R implementation of the function is composed of two steps. In the first one R creates the input data to be used for the C subroutine. In this first step R creates a time series from the input Y and it computes the sample mean $\bar{y}$ of Y as:

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

After this, it replaces values of Y with their depolarized value:

$$\hat{y}_i = y_i - \bar{y} \qquad \forall\, i \in [1, n]$$

At this point the C subroutine is invoked.

The C subroutine does the following steps:

1) for each lag $\tau \in [0, lag\_max]$ it computes the corresponding *Sample Autocovariance Function* $\hat{\gamma}(\tau)$ of the input time series (obtained using the normalization by n instead of $n - |\tau|$):

$$\hat{\gamma}_Y(\tau) = \frac{1}{n}\sum_{i=1}^{n-\tau}(y_i - \bar{y})(y_{i+\tau} - \bar{y}) = \frac{1}{n}\sum_{i=1}^{n-\tau}(\hat{y}_i \;\; \hat{y}_{i+\tau})$$

2) for each lag $\tau \in [0, lag\_max]$ it computes the corresponding *Sample ACF* $\hat{r}(\tau)$ of the input time series:

$$\hat{r}_Y(\tau) = \frac{\hat{\gamma}_Y(\tau)}{\hat{\gamma}_Y(0)} = \frac{\sum_{i=1}^{n-\tau}(\hat{y}_i \;\; \hat{y}_{i+\tau})}{\sum_{i=1}^{n}(\hat{y}_i)^2}$$

3) it returns the corresponding ACF $\hat{r}_Y(\tau)$ vector obtained

C. Profiling

We characterized R implementation of ACF over multivariate signals with increasing number of samples and dimensions, and reported the overall performances of the algorithm, in terms of execution time and quantity of memory allocated/deallocated. To visualize the profiling results we used Profvis a tool available from GitHub [25]. Profvis samples the state of the function call stack, by stopping the R interpreter at fixed time intervals (by default, every 10ms). Since R sampling profiler [26] results for each execution could be slightly different from one to another, we have executed the same profiling test functions multiple times and we have reported the average of the results. Tests have been executed on a Notebook with Intel Core i7-4710HQ CPU (2.50 GHz / 3.50 GHz, 4 core, 6 MB CACHE L3) and 4 GB DDR3L-1600 RAM (3,89 GB usable). To have a meaningful representation of how the performances of the algorithm scale with respect to the size of the data-set, we set changed value of lag_max from the default value: $lag\_max = 10 \cdot \log_{10}(nPts/nDim)$ with $nPts$ being the number of samples and $nDim$ the number of dimensions of the input signals, to: $lag\_max = (nPts/2)$

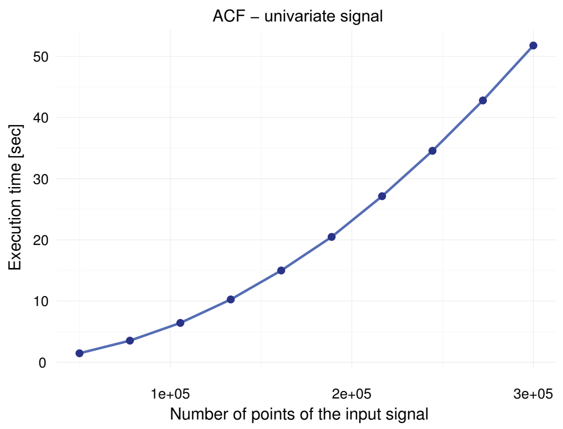


Fig. 1. Profiling results of ACF on CPU for univariate signals in terms of execution time. The increase in time complexity is quadratic with respect to the data-set size.

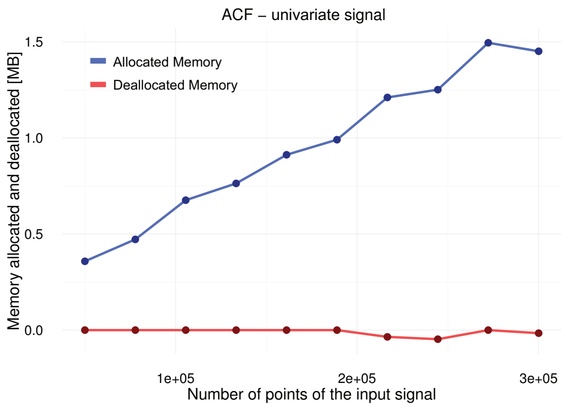


Fig. 2. Profiling results of ACF on CPU for univariate signals in terms of quantity of memory allocated/deallocated.

Using about half of the data-set size as maximum lag is a reasonable hypothesis, if the number of samples is considerable, as in our case. Figure 1 and Figure 2 show the average results for univariate signal with a number of observations ranging from 50K to 300K. The increase in time complexity is quadratic with respect to the dataset size. Figure 3 and Figure 4 show the average results on a dataset with a fixed amount of points (30K), and an increasing number of dimensions (from 1 to 10). Once again, the scaling of execution time is quadratic.

By inspecting the source code, we can compute the arithmetic intensity of the algorithm. We considered the number of memory accesses to floating point values, and the number of sums and multiplications performed on floating point values. Note: n is the number of points of the input signal, p is the number of dimensions, with $N = n \cdot p$, L is the maximum lag considered in the computation. The input of the algorithm will have approximatively size N, its output will have size $p^2 \cdot L$. The number of floating point memory accesses will be about $Lp^2(3 + 6n) + Lp^2 + p$ which can be asymptotically rewritten as $6Lp^2n$. The number of sums is $Lp^2n + 2Lp^2$ approximatively equal to Lp^2n. The number of multiplications is $Lp^2(1 + n) + 2Lp^2$, i.e. Lp^2n. Note that we considered of

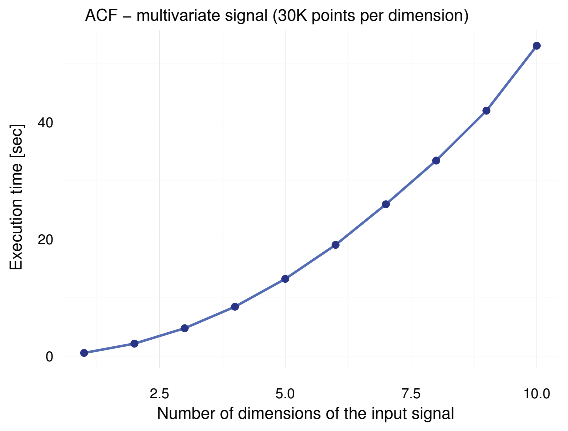


Fig. 3. Profiling results of ACF on CPU for multivariate signals in terms of execution time. The increase in time complexity is quadratic with respect to the data-set size.

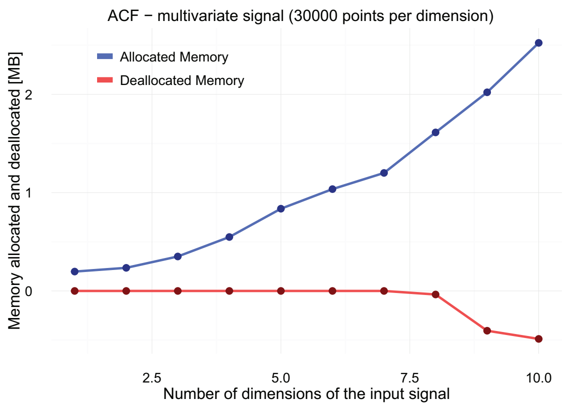


Fig. 4. Profiling results of ACF on CPU for multivariate signals in terms of quantity of memory allocated/deallocated.

equivalent complexity multiplications and divisions. p square roots are also performed: their number is, however, negligible compared to the ones of the other operations. As a result, the ratio between floating point operations (FLOPs) and memory accesses is:

$$\frac{FLOPs}{Memory\ accesses} = \frac{Lp^2n + Lp^2n}{6Lp^2n} = \frac{1}{3} = O(1)$$

while the arithmetical intensity, defined as ratio between the number of floating point operations and the input size is:

$$\frac{FLOPs}{InputSize} = \frac{Lp^2n + Lp^2n}{np} \approx Lp$$

If the maximum lag L is considerable, with an order similar to the number of points n, the arithmetical intensity becomes

$$Lp \approx np = N$$

From the results we obtained, it seems possible to improve the performances of ACF in a number of ways:

- *Dimensionality scaling*: in an n-dimensional signal, the computation of the correlation function between two of its dimensions is fully independent of the computation of the correlation functions of other dimensions. As a consequence, it is possible to evaluate in parallel the different correlation functions of the different components of the signal.
- *Lag*: taken the ACF of a univariate signal, the values of this function can also be computed independently from one to another. As an example, the value of the ACF $\hat{r}_Y(\tau = 0)$ can be computed separately from $\hat{r}_Y(\tau = 1)$.
- *Points*: in a given correlation function, for each value of lag, it is necessary to compute many vector products of the time series that compose the signal; the vector product is well suited to be accelerated in a number of ways, such as by making use of pipelined architectures or systolic arrays.

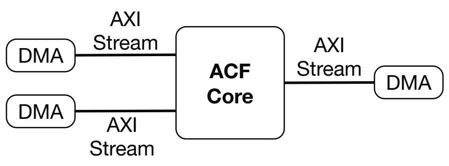


Fig. 5. Interface of the realized HW accelerator

IV. Hardware Accelerator

In this work we focused on the acceleration of the ACF of univariate signals. The solution presented here can be trivially extended to support multivariate signals, but this is not generally useful in practice. To support multivariate signals, it would suffice to re-use multiple times our implementation, with different signals as input. However, we believed more valuable to focus our efforts on optimizing as much as possible the base case of ACF and to put aside these extensions.

The first design challenge is that we need to take care on how our core accesses to the input data. In fact, the R implementation has the possibility to access each point of the signal 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. However on FPGAs random access to DDR on board is a costly operation that can take hundreds of clock cycles. For this reason, to achieve low memory access latencies, it is required to use registers (in the form of LUTs, look-up tables) and Block RAM (BRAM), which, however, are available in limited quantities. It is, in fact, possible to store a reasonable amount of data on the FPGA DDR (from 512MB to a few GB), if needed, but IP Cores can only access BRAMs and registers efficiently which can store data in the order of KB or at most MB.

However, if we look at the data access pattern of our ACF we can see that data is accessed sequentially from memory. Thanks to this, we can design our core to be able to accept data from a streaming interface, so that we do not need to store the whole input signal into local BRAM, but the core can simply access the data from the input streams when new data is needed. A controller is then needed to feed the input streams of the core with the data in the correct order. Being the access sequential, such controller can simply by a DMA which is instructed to copy data from one location to the input stream. Our implementation of the ACF needs 2 input streams, one for

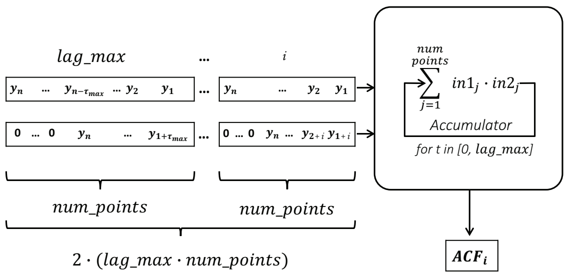
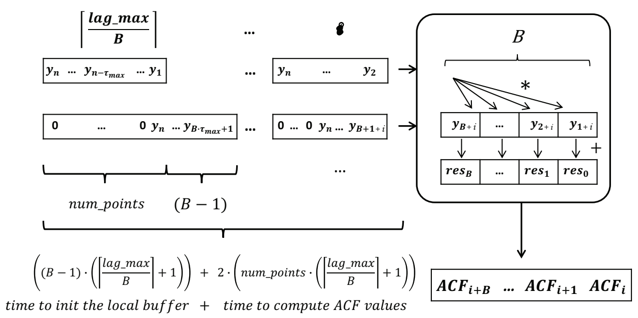


Fig. 6. Schema of the first FPGA implementation of the ACF.



$\left\lceil \frac{lag\_max}{B} \right\rceil$
B
*
res_B ... res_1 res_0
num_points
$(B-1)$
$\left((B-1)\cdot\left(\left\lceil\frac{lag\_max}{B}\right\rceil+1\right)\right) + 2\cdot\left(num\_points\cdot\left(\left\lceil\frac{lag\_max}{B}\right\rceil+1\right)\right)$
time to init the local buffer + time to compute ACF values
ACF_{i+B} ... ACF_{i+1} ACF_i

Fig. 7. Schema of the final FPGA implementation of the ACF.

the original signal, and one for the lagged one, and an output stream for collecting the output data, this structure defines the interface of our core, as presented in Figure 5. The streams are 32 bits wide to accommodate float data which is the datatype used for implementation. Vivado HLS easily allows to define streams and to define the protocol used to communicate with the external component by means of #pragma directives. We configured the core to use the standard AXI Stream protocol to communicate with Xilinx DMA cores that will be used to feed data to the core.

After defining the core interface we now describe how we implemented the computation of the ACF. At first, we focused on using as little resources as possible inside the IP Core, by exploiting a stream of input data with an appropriate structure: the idea is that to compute the *i-th* point of the ACF, we need to multiply each point of the signal with the other points, shifted by *i*. To do so without having to store the entire signal, or all the values of the ACF, inside the IP Core, we used the 2 input streams in the following way: one contains the signal (with size num_points), and another contains the signal shifted by an amount i. After reading the entire signal, we can output the i^{th} value of the ACF. This process is repeated lag_max times, the desired length of the output. Overall, we need to pass to the IP Cores $(2 \cdot num\_points \cdot lag\_max)$ values, as it is shown in Figure 6. By using hardware pipelining, it is possible to mask the cost if expensive multiply-and-accumulate operations, and read from the streams at every clock cycle. This implementation of the algorithm requires about $(num\_points \cdot lag\_max)$ operations, and its time complexity can be approximated to $O(n^2)$, while its spatial complexity is $O(1)$. Unfortunately, having to read $(2 \cdot num\_points \cdot lag\_max)$ becomes extremely expensive even for signals of moderate length: as an example, computing the ACF of a signal of size $50K$ with a lag_max of $25K$, and a clock frequency of $100MHz$, would require:

$$\frac{50K \cdot 25K}{100 \cdot 10^6} = 12.5$$

seconds, while in R this computation would take about 1.5 seconds.

It is clear that to achieve high performances it isn't enough to take advantage of hardware parallelism, but it is required to lower the amount of data transferred from the main memory. To do so, we decided to add two local buffers inside our IP Core, to store a small portion of the input signal and reduce the input streams size of a factor B (the local buffer size). The idea is that the product of a point x_i and the points in the range $[x_{i+lag},\ ...,\ x_{i+lag+B}]$ will be used to compute the ACF values in the range $[lag,\ ...,\ lag+B]$. To compute the value of ACF at position lag, with lag in the range $[1,\ ...,\ lag\_max]$, we need to multiply and accumulate every point in the original signal and in the signal shifted by a value lag. By using a local buffer that behaves as a shift register, it is possible to compute B values of ACF in parallel; their partial values are stored in the second local buffer, and are written to the output stream only when every pair $\{x_i,\ x_{i+lag}\}$ has been read from the streams. The input streams can be considered divided into blocks: each block has size num_points and allows to compute the ACF values in the range $[1,\ ...,\ lag\_max]$. The number of blocks is equal to lag_max/B. The blocks in the first input stream contain the full signal. The blocks in the second input stream contain the signal shifted by $B \cdot block\_number$. Figure 7 shows the delay caused by the initialization of the shift register at the beginning of each block. Once again, by employing hardware pipelining and loop unrolling, we were able to mask the latency of multiply-and-accumulate operations. The number of operations done by the algorithm is still $num\_points \cdot lag\_max$, approximated to quadratic time complexity $O(n^2)$, but now it is required to have two buffers of size B (which is still a $O(1)$ spatial complexity). However, 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. Our reference board, the Xilinx VC707, supported buffers of size 200, partitioned into blocks of size 50 so that it is possible to access 50 data in parallel realizing a SIMD architecture. 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, which can be easily done customizing C code given as input to Vivado HLS.

V. System Integration

After realizing the IP core, we need to perform the system integration phase to realize an HW architecture which allows us to use the realized accelerator from the host system. The solution proposed in this section is a first prototype of a system that can be used from a host device to perform the computation. The features that have to be made available by the HW architectures are: (i) the possibility to exchange data

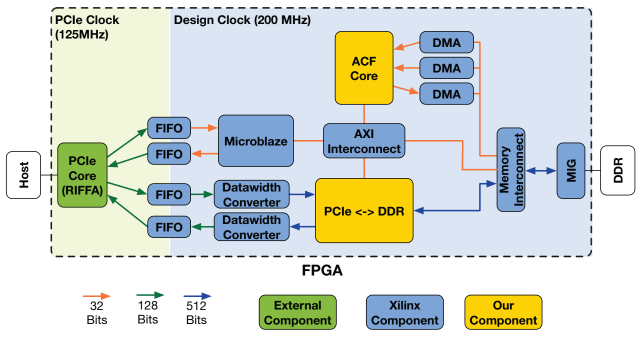


Fig. 8. HW Infrastructure to support communication with host.

via PCI-Express (PCIe) connection, (ii) the possibility to store data on DDR on board, (iii) the possibility to easily manage the allocation into the DDR memory, and (iv) the ability to control the HW accelerator in the design. Figure 8 illustrates the system we realized for satisfying the four points just mentioned.

The PCIe interface has been managed by using RIFFA, an open source solution from UCSD [27]. This solution, which is made available as pure Verilog, has been packaged in a Xilinx IP core and extended with the possibility to expose standard AXI Stream interfaces instead of the proprietary interface exposed by the original RIFFA core. The part of the design enclosing PCIe runs at 125MHz taking the clock from the PCIe slot, while the remaining of the system runs at the reference frequency generated by the MIG controller, which is the DDR controller made available by Xilinx. This reference frequency can be configured by the user, we used 200MHz in our design. In between these two clock domains we placed FIFO components both because we need buffering to store data coming from PCIe and because the Xilinx Data Stream FIFO component can be configured to act as a clock domain crossing component allowing the synchronization of the two asynchronous clocks present in the design.

The design allows data movement over PCIe to and from the DDR available on the board. Such data movement is possible thanks to a component we wrote in HLS that takes/writes data from/to the AXI Stream connections exposed by RIFFA and communicates with the MIG over an AXI Master connection. This core, thanks to the AXI Master interface, is able to issue read and write commands to the DDR and perform data movements. The core has also an AXI Slave configuration port where a controller can pass parameters such as the number of 512 bits words to move and the initial memory address. Upon receiving the start command the cores perform a sequential read or write from the provided initial memory address for the desired number of words. Core's inputs and outputs are 512 bits width since this datawidth allows to reach a high memory bandwidth; we measured a peak bandwidth of 11.7 GB/s on our Xilinx VC707 board.

The last two requirements, i.e. the possibility to easily perform memory management inside the FPGA and the possibility to control the HW core are possible thanks to a Microblaze instantiated in the design. The Microblaze communicates with the host system via PCIe, in fact, one AXI Stream coming from the RIFFA core is directly connected to the Microblaze stream interface. The Microblaze can then receive 128 bits instructions (in 4 32 bits words) from the host. Among these instructions, there are the request to allocate and deallocate a given number of bytes on DDR on-board (MALLOC and FREE). Upon receiving the command the Microblaze performs a *malloc()* or *free()* call and returns the result of the operation to the host, which now is aware of which part of the DDR is ready to be used for the computation. After allocating memory regions inside the FPGA the host can copy move data to/from the device by sending the Microblaze the proper instruction (DDR_WRITE or DDR_READ) and communicating the initial address and the number of bytes. The Microblaze takes care of configuring the core we implemented for moving data across PCIe to perform the requested operation. At this point, the host can check when the transfer is done with another instruction (CHECK_TRANSFER_DONE). Finally, the host can control the ACF core by issuing the RUN_CORE command and passing the needed parameters.

VI. Integration with R

The goal of building an interface between Higher Level Languages, such as R, Python and MATLAB, and an FPGA is to provide the user with algorithms that can be called like any other function. These algorithms will transparently make use of an FPGA implementation if certain criteria are met. As an example, when working with data of small size it might not be worth to use an FPGA, as the communication overheads would nullify any gain achieved by using hardware acceleration; in this case, the algorithm would fall back to a traditional CPU implementation. The user would still have to configure its FPGA appropriately (in terms of programming it with the right IP Core), but no specific knowledge of the board hardware and interfaces is required to use the algorithms from Higher Level Languages.

Generally, Higher Level Languages expose the possibility to call routines written in a more efficient and fast language such as C or C++. This is the case for Python, MATLAB and R. Boost-Python [15] is available to perform this operation and allows to easily move between Python and C++ NumPy [28] arrays which are the main datatype for a renowned Python mathematical library. Analogously in MATLAB one can use MEX files [13] to perform this MATLAB to C communication. The language target of this work, R, is no different from the other two and allows to connect R and C++ using the Rcpp package [16]. Rcpp allows to compile C/C++ code and build functions that can be called from R as if they were regular R functions. As R uses its own data types (e.g. arrays of floats use the class NumericVector) we have to convert the passed variable to standard C++ arrays before passing them to the FPGA, and vice-versa when returning the results to R. The conversion is handled by the R's C interface [29] which allows to cast the subtypes of defined SEXP data type to default C++ data types or R data types.

The passage from R to C++ is necessary, not only because C++ is faster than R in performing control operations, but also because RIFFA components do not directly expose drivers for R language, while it does for C. Once the data is ready on the C++ side, we can proceed with the communication with the FPGA exploiting the architecture presented in the previous

TABLE I. RESOURCE UTILIZATION BREAKDOWN AMONG DIFFERENT COMPONENTS OF THE PROPOSED ARCHITECTURE.

Component	LUTs	FFs	BRAMs 18K	DSPs
MIG	13568	15035	3	0
RIFFA	50409	65888	387	0
Microblaze	1749	2103	12	0
FIFOs	860	1523	229	0
Datawidth Converters	401	1564	0	0
Interconnects	8577	10463	160	0
DMAs	2395	3137	15	0
PCIe - DDR	1557	3707	0	0
ACF	83678	113458	0	503

section. In particular, for our case study the process happens in 5 steps:

1) Issuing of 2 MALLOC commands to reserve 2 memory regions inside the FPGA; one for the input time series and one for the ACF results;
2) Sending of a DDR_WRITE command to move input data onto the FPGA;
3) Running the computation via the RUN_CORE command;
4) Reading results back to the host using the DDR_READ command;
5) Releasing the memory regions allocated in the first step issuing 2 FREE commands.

Once these operations are completed the data is passed back to R by the Rcpp interface.

VII. RESULTS

This section presents the evaluation of this work by analyzing at first the resource usage of the devised HW architecture and then comparing the results of the proposed solution with the original R implementation.

Our solution has been implemented on a Xilinx VC707 board mounting a Xilinx Virtex7 xc7v456ff157-1 FPGA. This FPGA has been connected by means of a PCIe Gen2 connection to a host with an Intel Xeon W3530. The software solution has been tested on an Intel i7-4710HQ.

A. Resource Utilization

Table I reports a utilization breakdown of the different components in the HW architecture we designed. Looking at numbers we can see that the ACF core is the one using the most of the resources, but still the remaining of the design, which is used only for handling communication with the host, still occupies a relevant amount of resources. By the synthesis reports, we saw that the HW infrastructure, without the ACF core, uses: 26% of LUTs, 33% of FFs, and 39% of BRAMs. Most of the resource usage of the infrastructure is caused by RIFFA and the FIFOs used to buffer the data exchanged over PCIe. At the moment this is a limiting factor of our solution since it constrains a number of resources that can be used by the computational IP core. These numbers are still similar with a number of resources that are used by similar solutions, such as SDAccel [22, 23].

B. Performance

For evaluating the performance of our solution, we compared the results obtained implementing the ACF algorithm on the Xilinx VC707 board and the corresponding results obtained via native R library. Note that for signals with a low number of points the latency of transfer data through the PCIe can represent a bottleneck, so it results more convenient executing the computation on the CPU. To avoid these marginal cases, we analyzed signals with a number of points greater or equal to $50K$, which is a reasonable number in the context of data science and signal analysis. The tests were performed using univariate signals with increasing number of points, ranging from $50K$ to $1M$. The maximum lag considered, which coincides with the number of points of the ACF that were computed, is half of the number of points of the signal. From the results in Figure 9 it can be immediately seen that the FPGA massively outperforms the CPU as the signal size becomes bigger and bigger. To precisely quantify the improvements of our implementation over the default R one, we can compute the speedup, defined as:

$$Speedup = (\frac{Time_{CPU}}{Time_{FPGA}} - 1) \cdot 100$$

In Figure 10 it can be seen how the speedup grows very quickly up to about $300K$ points, then it slows down. This can be explained considering the overheads caused by the data transfer to the IP Core, that becomes negligible for large signals. Moreover, the ability to compute more lags in parallel can only reduce the complexity by a constant factor: this can be seen on large signals, for which the speedup value is almost constant with respect to the size of the input.

It should be noted that the speedup curve doesn't reach a stationary value in our analysis. As we know that the complexity of the considered algorithms is quadratic with respect to the number of points in the input, we performed a polynomial regression (of order 2) over the CPU and FPGA execution times and computed the speedup of the predicted execution times. In Figure 11 it can be seen that the speedup stops increasing with signals of more than 5 million points, with a theoretical speedup value of 700%. We also included

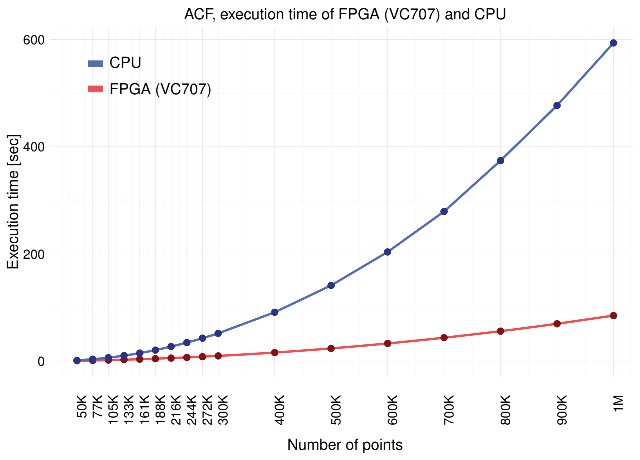


Fig. 9. Execution time of ACF tests for univariate signal with increasing number of points, on a Virtex-7 and on a CPU. The FPGA massively outperforms the CPU as the signal size becomes bigger and bigger.

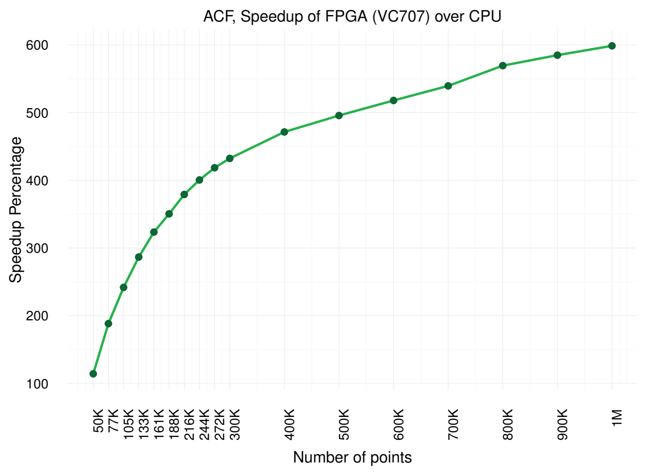


Fig. 10. Speedup percentage of a Virtex-7 over a CPU for a univariate signal with increasing number of points. The speedup grows very quickly up to about $300K$ points, then it slows down due to the overheads caused by the data transfer and the ability to compute a fixed number of ACF values in parallel.

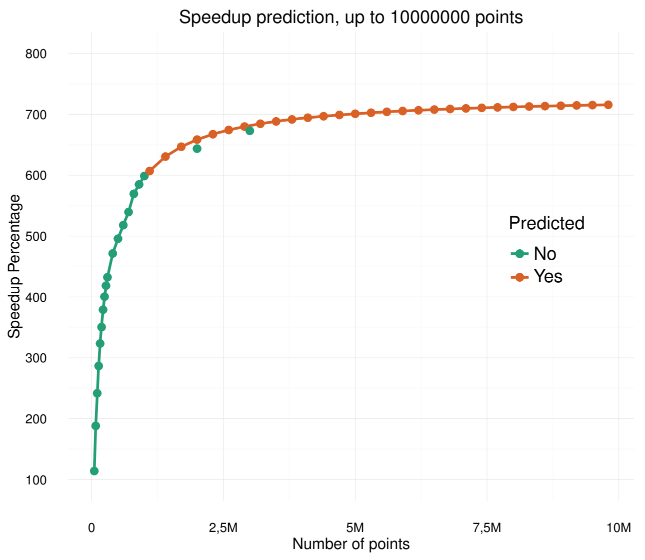


Fig. 11. Speedup prediction of a Virtex-7 over a CPU for univariate signal with increasing number of points. The speedup curve reach a stationary value with signals of more than 5 million points, with a theoretical speedup value of 700%.

the real speedup values for signals with $2M$ and $3M$ points, for comparison.

VIII. Conclusions

In this paper we presented how it is possible to exploit an FPGA to obtain accelerated version of algorithms that are widely used in the context of data science and signal analysis. As a case study, we accelerated a software implementation of the Autocorrelation Function (ACF) present in the default libraries of the R language. Furthermore, we presented how the realized accelerator can be integrated into a HW architecture that communicates with a host system via PCIe and allows the usage of the accelerator from a Higher Level Language. The solution presented in this work allows for the usage of the realized accelerator from the R language transparently to the final user which has simply to invoke the proper function to execute its analysis (ACF in our example) on the HW accelerator.

Future works will focus on multiple aspects. One of them will be the support of a wider range of algorithms and the development of the corresponding accelerators and the corresponding interfaces for Higher Level Languages. We will also broaden the support of languages including, for instance, MATLAB and Python. In doing these extensions we might also need to further abstract and revisit the interface with the HW system to be able to control different HW cores via the same interface. Furthermore, we also want to investigate two aspects on the HW side, the first one is the support of other PCIe interfaces as the one directly provided by Xilinx in the new versions of Vivado tools. Finally, we need to allow the HW to perform partial reconfiguration of the IP core performing the computation to allow the possibility to run different algorithms on the FPGA.

Acknowledgment

This work was supported by the European Commission in the context of the H2020 FETHPC EXTRA project (#671653).

References

[1] K. H. Tsoi and W. Luk, “Axel: a heterogeneous cluster with fpgas and gpus,” in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 115–124.

[2] NVIDIA Corporation, “Cuda parallel computing platform.” [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[3] F. Ferrandi, M. Novati, M. Morandi, M. D. Santambrogio, and D. Sciuto, “Dynamic reconfiguration: Core relocation via partial bitstreams filtering with minimal overhead,” in *2006 International Symposium on System-on-Chip*, Nov 2006, pp. 1–4.

[4] V. Rana, M. Santambrogio, and D. Sciuto, “Dynamic reconfigurability in embedded system design,” in *2007 IEEE International Symposium on Circuits and Systems*, May 2007, pp. 2734–2737.

[5] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[6] M. J. Flynn, “Some computer organizations and their effectiveness,” *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[7] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation,” *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[8] M. Baity-Jesi, R. Baos, A. Cruz, L. Fernandez, J. Gil-Narvion *et al.*, *An FPGA-based supercomputer for statistical physics: The weird case of Janus*. Springer New York, 3 2014, pp. 481–506.

[9] A. Parashar, M. Adler, M. Pellauer, and J. Emer, “Hybrid cpu/fpga performance models,” in *3rd Workshop on Architectural Research Prototyping (WARP 2008)*, 2008.

[10] M. D. Santambrogio, H. Hoffmann, J. Eastep, and A. Agarwal, “Enabling technologies for self-aware adaptive systems,”

in *2010 NASA/ESA Conference on Adaptive Hardware and Systems*, June 2010, pp. 149–156.

[11] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, “From opencl to high-performance hardware on fpgas,” in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012, pp. 531–534.

[12] Intel, “Intel fpga sdk for opencl.” [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf

[13] MathWorks, “Introducing mex files.” [Online]. Available: https://it.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html

[14] ——, “Mex file creation api.” [Online]. Available: https://it.mathworks.com/help/matlab/call-mex-files-1.html

[15] D. Abrahams and R. W. Grosse-Kunstleve, “Building hybrid systems with boost. python,” *CC Plus Plus Users Journal*, vol. 21, no. 7, pp. 29–36, 2003.

[16] CRAN, “Rcpp: Seamless r and c++ integration.” [Online]. Available: https://cran.r-project.org/web/packages/Rcpp/index.html

[17] H. M. Hussain, K. Benkrid, A. T. Erdogan, and H. Seker, “Highly parameterized k-means clustering on fpgas: Comparative results with gpps and gpus,” in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. IEEE, 2011, pp. 475–480.

[18] N. Scicluna and C.-S. Bouganis, “Fpga-based parallel dbscan architecture,” in *International Symposium on Applied Reconfigurable Computing*. Springer, 2014, pp. 1–12.

[19] X. Wang and X. Wang, “Fpga based parallel architectures for normalized cross-correlation,” in *2009 First International Conference on Information Science and Engineering*, Dec 2009, pp. 225–229.

[20] B. Miao, R. Zane, and D. Maksimovic, “A modified cross-correlation method for system identification of power converters with digital control,” in *2004 IEEE 35th Annual Power Electronics Specialists Conference (IEEE Cat. No.04CH37551)*, vol. 5, June 2004, pp. 3728–3733 Vol.5.

[21] J. E. Stone, D. Gohara, and G. Shi, “Opencl: A parallel programming standard for heterogeneous computing systems,” *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[22] J. Fifield, R. Keryell, H. Ratigner, H. Styles, and J. Wu, “Optimizing opencl applications on xilinx fpga,” in *Proceedings of the 4th International Workshop on OpenCL*. ACM, 2016, p. 5.

[23] G. Guidi, E. Reggiani, L. Di Tucci, G. Durelli, M. Blott, and M. D. Santambrogio, “On how to improve fpga-based systems design productivity via sdaccel,” in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 247–252.

[24] NIST/SEMATECH, “e-handbook of statistical methods.” [Online]. Available: http://www.itl.nist.gov/div898/handbook/eda/section3/eda35c.htm

[25] Winston Chang, “Profvis.” [Online]. Available: https://github.com/rstudio/profvis

[26] ——, “Profvis intro.” [Online]. Available: https://rpubs.com/wch/123888

[27] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, “Riffa 2.1: A reusable integration framework for fpga accelerators,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, p. 22, 2015.

[28] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, “The numpy array: a structure for efficient numerical computation,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[29] Advanced R by Hadley Wickham, “Rs c interface.” [Online]. Available: http://adv-r.had.co.nz/C-interface.html