Towards Benchmarking GNSS Algorithms on FPGA using SyDR

Antoine Grenier*, Hans Jakob Damsgaard*, Jie Lei†, Enrique S. Quintana-Ortí†,
Aleksandr Ometov*, Elena Simona Lohan*, Jari Nurmi*
**Electrical Engineering Unit, Tampere University*, Finland, {first-names.lastname}@tuni.fi
†*Parallel Architectures Group, Universitat Politècnica de València*, Spain, {jlei, quintana}@disca.upv.es

***Abstract*—Global Navigation Satellite System (GNSS) is widely used today for both positioning and timing purposes. Many distinct receiver chips are available off-the-shelf, each tailored to match various applications' requirements. Being implemented as Application-Specific Integrated Circuits, these chips provide good performance and low energy consumption but must be treated as "black boxes" by customers. This prevents modification, research in GNSS processing chain enhancement (e.g., application of Approximate Computing techniques), and design-space exploration for finding the optimal receiver implementation per each use case. In this paper, we review the development of *SyDR*, an open-source Software-Defined Radio oriented towards benchmarking of GNSS algorithms. Specifically, our goal is to integrate certain components of the GNSS processing chain in a Field-Programmable Gate Array and manage their operation with a Python program using the Xilinx PYNQ flow. We present the early steps of converting parts of SyDR to C, which will be later converted to Hardware Description Language descriptions using High-Level Synthesis. We demonstrate successful conversion of the tracking process and discuss benefits and drawbacks arising thereof, before outlining next steps in preparation for hardware implementation.**

***Index Terms*—Benchmarking, Computational complexity, Field-programmable gate array (FPGA), Global Navigation Satellite System (GNSS), Open-source software**

I. Introduction

Today, satellite positioning systems or Global Navigation Satellite System (GNSS) are widely used in any type of electronics for both positioning and timing purposes [1]. A variety of GNSS receiver chips is available off-the-shelf for integration in embedded devices and for multiple applications, e.g., high-precision, autonomous driving, low-power, etc. To maximize efficiency and energy consumption, they are implemented as Application-Specific Integrated Circuit (ASIC)s. Such integrated circuits can be considered "black boxes", as they allow neither inspection nor modification of their internals. While this is perfect from an industrial point-of-view, it is not suitable for research purposes, as it is impossible to explore new processing algorithms on this kind of a platform.

GNSS receivers remain some of the most energy-consuming sensors [2]. Their power consumption can come from various factors: user requirements, operating environment, electronics, etc. All of these influence the software implemented inside the receiver. Determining the optimal software implementation for maximum performance and minimum computational burden is difficult and different for every application case. As these receivers are expected to be integrated into an ever-increasing number of battery-powered embedded devices, understanding their performance-energy trade-off becomes more interesting.

In the past, software-defined GNSS receivers have been used extensively for testing new Digital Signal Processing (DSP) algorithms. Several of these tools are available in open-source [3]–[7]. However, they all suffer from at least one of these issues: lack of maintenance, limited-to-no hardware integration, architecture differing from an actual receiver, or not being designed for benchmarking purposes. This final point is of particular interest for us, as our goal is to explore the GNSS processing chain to find points where energy consumption could be reduced, for example through approximation.

Approximation is at the core of Approximate Computing (AxC), denoting a domain in which small, constrained computational errors are traded off for reduced execution time, circuit area or power, and thereby energy consumption [8]. It is crucial to note that AxC can be applied only in applications that are resilient to such errors, i.e., those whose outputs remain acceptable despite some reduced precision. Examples are in Machine Learning (ML) and in DSP algorithms [9]. We note a lack of work using AxC in GNSS processing and highlight it as a promising direction for future research.

A. Related Work

As described above, most GNSS receivers are implemented in hardware rather than in software to achieve satisfactory performance. Benchmarking GNSS algorithms in hardware can allow for additional metric estimation and enhance their comparability. Producing ASICs for this purpose would be unreasonably time-demanding and costly. Field-Programmable Gate Arrays (FPGAs) are a suitable alternative platform, allowing for reconfiguration with little effort.

In this direction, Xilinx (now acquired by AMD) has proposed a hardware-software co-design workflow called *PYNQ*, aiming to accelerate Python-based algorithms by porting computationally expensive parts to an FPGA on their Zynq chips [10]. This workflow has swiftly gained appeal in a variety of research domains. Especially in the ML community, the computation-intensive design space exploration of convolutional neural networks can be sped up by scalable PYNQ

The authors gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation Programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, http://www.apropos-itn.eu/).

implementation [11]. Furthermore, support for low-precision number formats enables using AxC to increase neural network throughput [12].

A communication engineer can also use PYNQ to accelerate an Software-Defined Radio (SDR)'s Fast Fourier Transform (FFT) and signal modulation operations, as illustrated by Goldsmith *et al.* in [13]. They propose an open-source SDR to provide software control of radio functionality as well as real-time radio signal visualizations. Similar research was also conducted on radar warning receivers in [14], for which the adoption of PYNQ is shown to reduce the overall system execution time by a factor of 2 over executing the algorithms in an embedded Central Processing Unit (CPU).

B. Previous Work and Contributions

In [15], we introduced the concept of our SDR "SyDR" (previously named "pyGNSS-SDR"), designed for GNSS processing. We analyzed existing GNSS SDRs and the benefits of our architecture. The first objective was to develop a receiver in a high-level language, in our case Python. The software is provided as open-source[1] and can be used as a playground to evaluate the performance of different algorithms. As Python is known to have significant runtime overheads [16], we already foresaw its conversion to a lower-level language (e.g., C or Rust) and its implementation on hardware such as an FPGA.

In this paper, we explore the progressive conversion of SyDR to a low-level language and later to hardware. In order to avoid redundant work, we initially only perform partial conversion. We have chosen the PYNQ environment, developed by Xilinx to enable Python and FPGA interactions on their Zynq-based development boards. Advantages of going from software to hardware are multiple: 1) enabling AxC only accessible at the hardware level [9]; 2) more realistic energy consumption estimates; and 3) processing acceleration. A partial conversion should provide the best of both worlds, that is, efficient processing for computationally intensive tasks, and benchmarking tasks performed in a rich and high-level environment. We summarize our contributions in here as follows:

- **C1** Highlighting the compute-heavy tasks in GNSS processing;
- **C2** Reviewing the PYNQ environment;
- **C3** Devising a methodology for going from Python to hardware;
- **C4** Enhancing SyDR for the mixed Python & C integration.

The goal of this paper is thus to introduce our conversion workflow, going from a pure Python implementation to a partial hardware implementation, using the PYNQ environment.

C. Paper structure

The rest of the paper is organized as follows. We start by reviewing the GNSS receiver architecture implemented in SyDR to highlight the components to implement in hardware in Section II. The conversion from software to hardware, as well as the details of the PYNQ environment, are detailed in Section III. As this conversion is still ongoing, we provide intermediate results and outline future work in Section IV. Finally, Section V summarizes our work.

[1]See https://github.com/aproposorg/sydr.

II. Background

A. GNSS processing chain

Signal reception in communication can be separated into two parts: analog and digital processing. The first one is performed on analog signals, i.e., pre-digitization. It accounts for signal reception by the antenna, and filters/amplifiers in the Radio Frequency (RF) front-end. Once the signal is digitized, DSP techniques can be applied. In this project, we focus only on the DSP part. We do not plan any RF front-end development. Instead, we focus on implementing a benchmarking chain using pre-recorded RF signals, preventing the need for real-time acquisition.

In a typical GNSS DSP processing chain, a receiver will undergo three essential stages described next, to reconstruct the pseudoranges and obtain a position:

Acquisition is performed when a new GNSS signal needs to be tracked. At that point, the exact parameters of the signal, i.e., frequency and ranging-code shifts, are unknown and need to be estimated. The acquisition performs a coarse estimation of these two parameters, so that a replica of the signal can be generated. If the acquisition stage fails to retrieve the signal, the receiver does not proceed to the tracking stage.

Tracking represents the fine-tuning process to align precisely the code and the frequency of the signal replica with the incoming signal. Once acquired, a signal should be continuously tracked to remove the code and produce the necessary measurements for pseudorange reconstruction. The process remains at this stage until the end of the positioning scenario. Under certain conditions, e.g., receiver's high dynamics, the process may lose the tracking, causing it to revert to the acquisition procedure.

Decoding occurs in parallel with tracking. It involves recovering the bits of the navigation message from the code-free signal and decoding them into essential parameters, including the transmission time in the form of Time of Week, satellites' orbital details, atmospheric models, etc. This information combined with the measurements from the tracking loops are used for pseudorange reconstruction and positioning computations.

Acquisition is often highlighted as the most computationally complex, as it involves testing many combinations of frequency and code shifts, resulting in thousands of correlation operations. Yet, as long as the receiver does not lose track of the signal later on, the acquisition should be performed only once. In contrast, tracking requires only a few correlation operations, but is performed frequently (e.g., at a rate of 100-1000 Hz), and continuously until the end of the receiver activity. Compared with the navigation algorithms, acquisition and tracking account for most of the energy consumption [17].

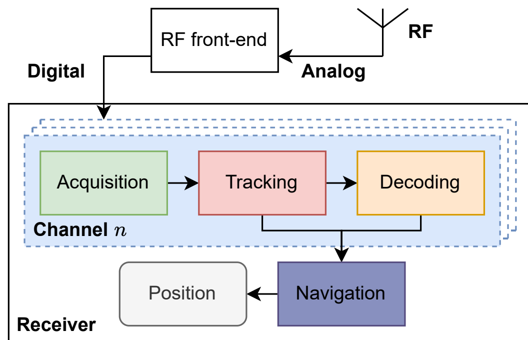

Fig. 1. Typical GNSS processing chain, as implemented in SyDR.

As each signal can be tracked in parallel, DSP stages are grouped in *channels*. A receiver with N channels can track up to N satellites concurrently. Figure 1 summarizes the processing chain explained above, with the essential components implemented in a GNSS receiver.

B. SyDR, an Open-Source Benchmark Platform

The SyDR software follows the architecture described above. In [15], we defined the state machine logic of the *receiver* and *channels*. Each channel is responsible for acquiring and tracking a specific satellite requested by the receiver.

Channels can be viewed as independent processes that require new RF data as inputs and provide updates on their current tracking state to the receiver. In SyDR, a channel operates in a "starving" manner: all the data available to it will be processed according to the state-machine logic, until there is no more data to process. The channel then waits for more data to be passed to it by the receiver. Consequently, the receiver evolves symmetrically, repeatedly passing a batch of data to the channels and waiting for them all to return their results. This logic allows asynchronous operation of the channels and the receiver, regardless of the processing times, while synchronizing the measurements in the receiver.

The channels perform the DSP computations, highlighted previously to be most energy consuming. This is due to the sampling frequency and the amount of RF data to be processed. We have confirmed such computational load during our tests with SyDR.Given the amount of computations, AxC techniques could open a new door to low-power GNSS processing, providing energy consumption reductions. Although most AxC techniques can be applied or emulated in software, many of them can only be applied efficiently in hardware [9]. Moreover, as these techniques often imply consequent accuracy/precision reduction in computations, their effect on the signal tracking and positioning needs to be evaluated.

Theoretical evaluation of the effect of AxC techniques on the processing chain is possible and under study in our current experiments using SyDR. Yet, results thereof are not included in this paper, as we focus here on the definition of the hardware conversion workflow.

C. Towards Hardware Acceleration

In its current implementation, the channels of SyDR are executed as independent processes using the standard `multiprocessing` library. The communication between the receiver and its channels is passed through *pipes* that provide new data and retrieve the channels' results. This encapsulation was performed to enable parallel processing in the software. In the foreseen objective to move all the processes of the channels to hardware, it also allows for an easier transition between the software and hardware definitions of the channels.

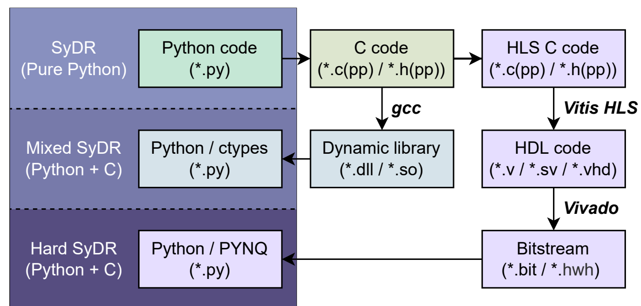

Fig. 2. Conversion workflow for SyDR from pure Python to partial hardware implementation enabled by PYNQ and HLS.

III. From Software to Hardware

As explained before, our aim is to run SyDR closer to the hardware by developing a partial implementation on an FPGA. This is done to obtain better complexity estimates and directly apply AxC techniques at the hardware level. For this purpose, we have decided to use the PYNQ tool flow that neatly integrates Python with hardware designs implemented on FPGA in Xilinx Zynq chips [10], [18].

A. Interfacing Python with Hardware

Zynq chips are logically split into two parts: the Processing System (PS) comprising general-purpose (ARM) CPU cores, caches, I/O devices, etc.; and the Programmable Logic (PL) comprising an FPGA [18]. The PYNQ flow allows a developer to reason about data movement between the PS and the PL, and dynamically implement different hardware designs (denoted by *overlays*) from Python code running in the PS. To be implemented in the FPGA, the designs must be *synthesized* into *bitstreams* using Xilinx Vivado. This requires them to be written in a HDL such as VHDL or (System)Verilog. As these languages are distinct from any high-level programming language and fundamentally evaluated in parallel rather than sequentially, this involves a step of design conversion. As SyDR is developed in Python, this conversion can be tackled in two ways: 1) directly from Python to HDL, or 2) from Python to a lower level language like C or C++, and from thereon to HDL using HLS.

The first option, while being more direct, generally requires long development cycles to implement and debug the algorithms in HDL. To avoid that, we pursue the second option, converting Python code to C, and using Xilinx Vitis HLS to generate HDL code [19]. We do so for four reasons: first, it significantly reduces our development time. Second, the HLS flow automatically configures the (Advanced eXtensible Interface (AXI)-based) interfaces needed between the PS and

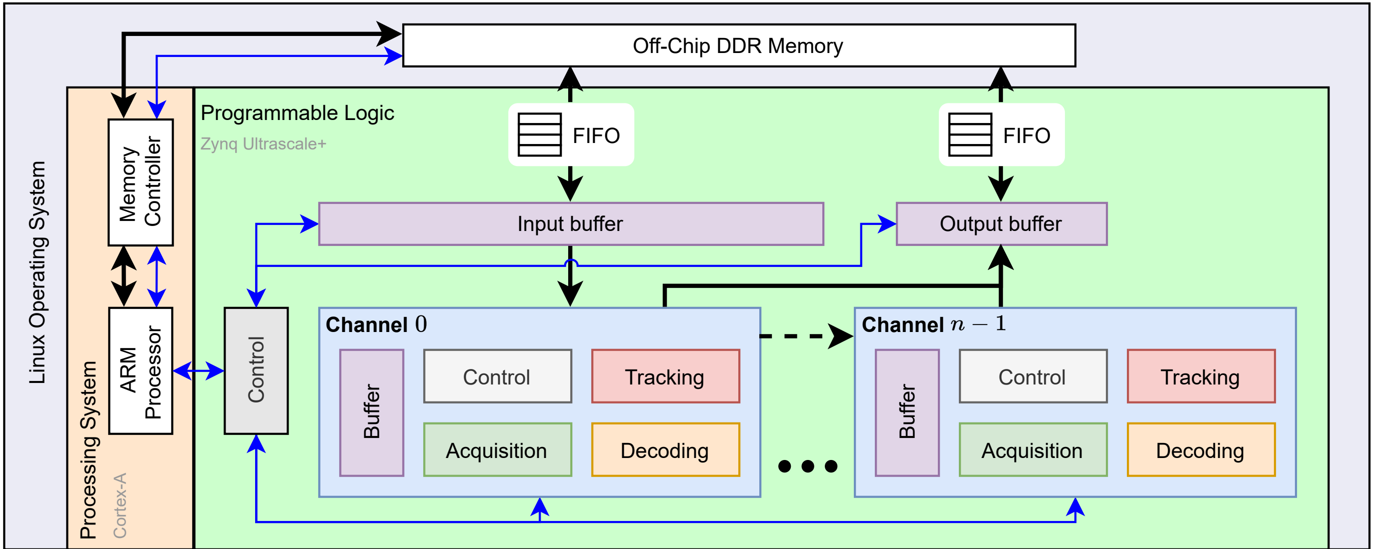

Fig. 3. Overview of the hardware architecture we aim for with SyDR accelerated on a Zynq FPGA.

PL. Third, it enables us to test our algorithms independent of an FPGA board using one of Python's libraries for interfacing external functions, e.g., `ctypes`. And last, converting the code to C allows us to apply and explore compiler-enabled AxC techniques, e.g., precision tuning [20], when compiling the code. It might also speed up processing compared to the pure Python implementation, described further in Section IV.

When we have converted the functions that need acceleration to C, we first generate their HDL descriptions using Vitis HLS. The HLS flow is aware of the parallel nature, delays, potential race conditions, etc. that are present in hardware designs and produces a design that exploits or avoids these. The developer can affect parts of this flow to determine dataflow, pipelining, and targeted initiation intervals of individual functions, loops, and other C constructs. Next, we synthesize the resulting design into a bitstream using Vivado. Then we implement a PYNQ-compatible overlay in the Python code that takes this bitstream as a parameter [10]. This overlay simplifies communication between the Python code and the design. The complete workflow with indication of file types is summarized in Figure 2.

B. Conversion from Python to C

As highlighted in Section II, we want to keep the majority of the code in Python and only move compute-heavy tasks to the PL. Thus, at first, we only consider the code related to channels (see Section II) for conversion, as they involve performing FFT and correlation operations between complex-valued measurement streams; operations that can be parallelized in lower-level languages and in hardware. Python, being a versatile language, supports directly loading a dynamic library file, i.e., `.dll` or `.so`, within a script using the `ctypes` library. This allows us to debug our C functions quickly by comparing them to their Python counterparts.

If implemented properly, the C code simply needs to be annotated with HLS-related pragmas to ensure proper conversion by Vitis HLS. Examples of these include forcing top-level I/O to be implemented using AXI interfaces, as well as various architectural details that affect the design's latency and throughput. The former is crucial to ensure good performance in memory transfers. The resulting HDL code must be synthesized into a *bitstream* file by Vivado before the PYNQ flow can load it into PL, as outlined in Figure 2.

C. Envisioned Accelerator Design

The aim of our conversion is an architecture resembling that shown in the PL section of Figure 3. Following SyDR's structure, the design will include buffers for storing measurements and channel outputs, and any number of channels. The PYNQ flow supports a variety of protocols for managing data transfers across the AXI-based interface. As the PL has direct access to the PS' memory subsystem, one solution is to allocate buffers in memory and pass addresses to these to the PL, that can later access the memory-based buffers. While this is flexible, it is more efficient to use Direct Memory Access (DMA)-based transfers, especially for large, contiguous chunks of memory like the measurement data used in SyDR. Doing so may also simplify the shown control logic. The input buffer can be circular and sized according to SyDR's needs, e.g., storing $4\,ms$ of data, while the output buffer can store just one response from each channel to be returned once their processing is finished. This event can, for example, be flagged with an interrupt.

As described in Section II-A, the channels are independent but identical, i.e., they will track different satellites, operating initially from the same point in the measurement stream till they have acquired a signal and its offset. From a hardware designer's point-of-view, this leads to two main observations: 1) the similarity of the channels suggests opportunities for operator (or function) reuse to avoid unnecessary duplication, and 2) the expectedly irregular memory accesses of the channels may render using a single input buffer infeasible. The

former requires some design space exploration to identify the best area-latency trade-off, taking the channels' concurrency into account. The latter may be solved by adding extra control logic to the top-level.

Controlling the accelerator is possible over another AXI-based interface that the HLS flow automatically generates. With it, the registers in the design become memory-mapped and, thus, available to the developer via the PYNQ flow. The overlay will implement methods that interact with these, for example, to set which satellite each channel should track.

Unlike streaming accelerators such as [12], our design is stateful and to some degree free-running. That is, following the processing flow of SyDR, roughly one millisecond of data will be transferred to the accelerator at a time. Once the input buffer has enough data to perform the DSP operation, data processing starts and one collective response will be returned. Depending on the tracking frequency, the response can be sent at different intervals, e.g., for every $1\,ms$ of data. Maintaining the state of the channels and the buffers meanwhile is crucial to ensure progression from acquisition to tracking, etc. Using a DMA and interrupts allows us to decouple the Python code in the PS from the accelerator in the PL, such that they can proceed concurrently.

For initial validation of our system, we will use the default data types of the `numpy` library, i.e., C's `long` and `double` types. Yet, as floating-point arithmetic is costly to implement in FPGA, we expect to replace them with suitably sized fixed-point alternatives, available in Xilinx's arbitrary precision libraries [19]. Wide integers are less problematic to implement but may cause unnecessary strain to the PL's memory interface. Finding the optimal trade-off here will involve a design space exploration similar to the aforementioned.

IV. Intermediate Results

The conversion from software to hardware is ongoing. We have focused on converting the acquisition and tracking operations first, as they represent most of the computations in the DSP stage. As explained in Section III, we have converted the functions to a compiled C library, which is loaded into Python using the `ctypes` library. This implementation allows us to debug the C code directly in SyDR's environment, thanks to SyDR's modular Application Programming Interface.

An overview of the positioning results is presented in Figures 4 and 5. Table II provides several statistics of the positioning precision and accuracy of the software. Currently, the results available are worse than the ones presented by other open-source SDRs [21], [22]. Yet, these results are in a similar order of magnitude for a single-frequency Standard Point Positioning solution. Note that only six satellites were used for the computation, due to current software limitations. Future software updates should solve this issue and provide enhanced capacities for, e.g., tracking more than six satellites or other constellations, and supporting alternative algorithms. Nevertheless, successful positioning is achieved by the software in a mixed Python & C configuration, allowing us to move forward with the hardware implementation.

To further assess the mixed SyDR implementation, we have compared its relative processing times with those of the pure Python implementation. The two processing runs were performed using Python 3.8 on an 8-core Intel i7-10700K CPU with 32 GB of RAM over the same scenario as in [15], whose parameters are listed in Table I, producing more than thirty thousand execution time measurements for the two tracking implementations. Ignoring any potential interference from the operating system, this results in a reasonably representative dataset for execution time analysis. As shown in Table III, the C functions do not speed up processing. This is likely due to the very crude data type selection and memory copying occuring between Python and C interactions. More refined data type conversion should result in a speedup in future implementations. Nevertheless, acceleration is not the main goal of this conversion, as we aim for the subsequent hardware implementation.

Reference dataset		
Date		2021.11.30, ∼ 8:40 (UTC)
Location		TAU Rooftop (open-sky)
Dynamic		Static
Antenna		Novatel GPS-703-GGG
Instruments	RF Logger	NI USRP-2953R
	Clock	Spectracom GSG-6
Frequency	Center	1575.42 MHz (L1)
	Bandwidth	120 MHz
	Sampling frequency	10 MHz
	Quantization	8 bits
	I/Q	Complex
Acquisition		
Method		PCPS
Doppler Range		± 5000 Hz, 100 Hz step
Integration		5 ms coherent, 10 ms non-coherent
Threshold		Ratio two highest peak, 1.5
Tracking		
Method		Early Prompt Late
Correlator spacing		-0.5 / 0 / 0.5
DLL	PDI	0.001
	Dumping ratio	0.7
	Noise bandwidth	2.0 Hz
	Loop gain	1.0
PLL	PDI	0.001
	Dumping ratio	0.7
	Noise bandwidth	25.0 Hz
	Loop gain	0.25
Navigation		
Method (pseudorange)		Common reception time
Method (position)		Least-Squares Estimation
Satellite used		GPS PRN 2, 3, 4, 6, 9, 29
Measurements		Pseudoranges L1 C/A only
Frequency		1 Hz

TABLE I
SIMULATION PARAMETERS OF THE INTERMEDIATE RESULTS.

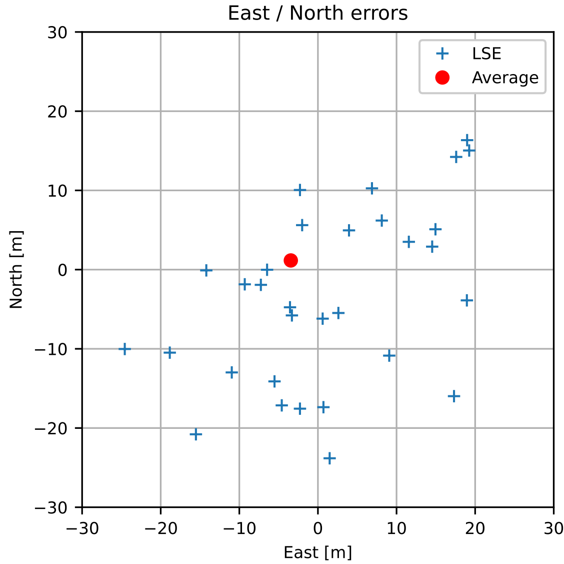

Fig. 4. East/North errors relative to the reference coordinates

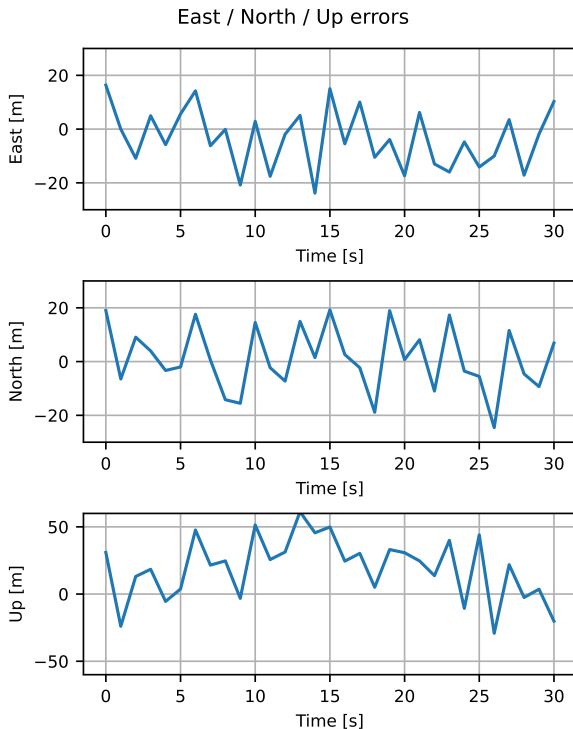

Fig. 5. East/North/Up errors over time relative to the reference coordinates.

Errors	East [m]	North [m]	Up [m]	2D (E/N) [m]	3D [m]
Average	-3.45	1.16	19.41	9.52	30.93
STD (1σ)	10.59	11.69	22.82	6.30	6.78
Maximum	23.82	24.57	61.20	23.81	26.54
Minimum	0.02	0.60	2.54	0.02	5.94

TABLE II
STATISTICS OF POSITIONING ERROR IN THE LOCAL FRAME (ENU)

A. Future Work

The current code only has the tracking loops converted to C in the mixed SyDR implementation. For enhanced performance, we need to move further in the processing chain and fully convert all the channels' processing steps outlined in Section II-A. Converting the whole channel structure is necessary to encapsulate the processing to construct it in hardware. In parallel, we will mostly focus on developing the conversion workflow from Python to PYNQ, described in Section III. Once the mixed SyDR implementation with channel conversion is completed, and our conversion workflow has been tested successfully, we will move forward with the transfer of the channels to hardware. To implement these, we will introduce a top-level module called the "*ChannelManager*" that incorporates control logic and buffering needed for the processing, as well as instantiates any requested number of channels. Introducing this top-level module allows us to minimize interface code between Python and the FPGA.

PRN	SyDR (Python)		Mixed SyDR (Python + C)		
	Avg. [μs]	STD [μs]	Avg. [μs]	STD [μs]	Speedup
G02	1124.647	355.428	1193.357	265.319	0.94
G03	1106.482	273.059	1199.425	269.308	0.92
G04	1166.107	400.813	1193.162	275.642	0.98
G06	1101.146	359.601	1172.152	264.846	0.94
G09	1139.108	365.339	1197.333	268.851	0.95
G29	1139.633	361.928	1189.150	261.477	0.96

TABLE III
STATISTICS OF PROCESS TIME BETWEEN PURE PYTHON AND PYTHON+C IMPLEMENTATION.

As we aim to participate in the Xilinx Open Hardware Competition 2023, we foresee completing the workflow implementation over the next months. Additional performance results will, thus, be covered in the conference presentation.

V. CONCLUSION

In this paper, we have reviewed a typical GNSS receiver's architecture and compared it to the SyDR software, an open-source SDR for GNSS algorithm benchmarking. We have also shown that due to their encapsulation and compute-intensive tasks, channels are ideal candidates for AxC and hardware acceleration.

In parallel, the PYNQ environment available on Xilinx Zynq-based boards allows for partially migrating SyDR's code from Python to hardware logic. This enables direct implementation of AxC techniques in hardware, while moving the code closer to the hardware layer and, thus, possibly enabling energy estimation. We have proposed a conversion workflow to evolve from pure Python implementation to the Zynq platform. Details of the interfaces between the conversion tools have been theoretically defined and explained. We provided select intermediate results, showing the current progress in code conversion. Finally, the future developments foreseen have been reviewed and exposed, to progress towards a hardware implementation of the SyDR software.

REFERENCES

[1] A. Grenier, E. S. Lohan, A. Ometov, and J. Nurmi, "A Survey on Low-Power GNSS," *IEEE Communications Surveys & Tutorials*, 2023.

[2] S. Narayana, R. V. Prasad, V. Rao, L. Mottola, and T. V. Prabhakar, "Hummingbird: Energy Efficient GPS Receiver for Small Satellites," in *Proc. of the 26th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3372224.3380886

[3] D. M. Akos, *A Software Radio Approach to Global Navigation Satellite System Receiver Design*. Ohio University, 1997.

[4] K. Borre, D. M. Akos, N. Bertelsen, P. Rinder, and S. H. Jensen, *A Software-Defined GPS and Galileo Receiver, a Single Frequency Approach*. Birkhäuser, 2007.

[5] C. Fernández–Prades, J. Arribas, P. Closas, C. Avilés, and L. Esteve, "GNSS-SDR: An Open Source Tool for Researchers and Developers," in *Proc. of the 24th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS 2011)*, Portland, OR, Sept. 2011, pp. 780–794.

[6] Finnish Geodetic Institute, "The FGI-GSRx Software Defined GNSS Receiver goes Open Source," [Online] https://www.maanmittauslaitos.fi/en/topical_issues/fgi-gsrx-software-defined-gnss-receiver-goes-open-source, 2022.

[7] K. Borre, I. Fernández-Hernández, J. A. López-Salcedo, and M. Z. H. Bhuiyan, *GNSS Software Receivers*. Cambridge University Press, 2022.

[8] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[9] H. J. Damsgaard, A. Ometov, and J. Nurmi, "Approximation Opportunities in Edge Computing Hardware: A Systematic Literature Review," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–49, 2022.

[10] Xilinx/AMD, "Pynq," https://github.com/Xilinx/Pynq, 2022.

[11] F. Kästner, B. Janßen, F. Kautz, M. Hübner, and G. Corradi, "Hardware/Software Codesign for Convolutional Neural Networks Exploiting Dynamic Partial Reconfiguration on PYNQ," in *Proc. of IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 154–161.

[12] H. Kim and K. Choi, "The Implementation of a Power Efficient BCNN-Based Object Detection Acceleration on a Xilinx FPGA-SoC," in *Proc. of International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2019, pp. 240–243.

[13] J. Goldsmith, C. Ramsay, D. Northcote, K. W. Barlee, L. H. Crockett, and R. W. Stewart, "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework," *IEEE Access*, vol. 8, pp. 129 012–129 031, 2020.

[14] J. Juliano, J. Lin, A. Erdogan, and K. George, "MPSoC FPGA-Based Radar Warning Receiver," in *Proc. of IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2021, pp. 1–6.

[15] A. Grenier, E. S. Lohan, A. Ometov, and J. Nurmi, "An Open-Source Software-Defined Receiver for GNSS Algorithms Benchmarking," in *Proc. of 14th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE, 2022, pp. 31–38.

[16] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking Programming Languages by Energy Efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021.

[17] K. Chen, G. Tan, and M. Lu, "Improving the Energy Performance of GPS Receivers for Location Tracking Applications," in *Proc. of IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017, pp. 85–90.

[18] Xilinx/AMD, "Zynq UltraScale+ MPSoC Data Sheet: Overview," [Online] https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview, 2021.

[19] Xilinx, "Vivado Design Suite User Guide - High Level Synthesis," [Online] https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis, 2021.

[20] S. Cherubin, D. Cattaneo, M. Chiari, and G. Agosta, "Dynamic Precision Autotuning with TAFFO," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 2, pp. 1–26, 2020.

[21] S. Islam, M. Z. H. Bhuiyan, N. Linty, and S. Thombre, "GPS L5 Software Receiver Implementation in FGI-GSRx," *International Union of Radio Science (URSI): Tampere, Finland*, 2019.

[22] U. Robustelli, M. Cutugno, J. Paziewski, and G. Pugliano, "GNSS-SDR Pseudorange Quality and Single Point Positioning Performance Assessment," *Applied Geomatics*, pp. 1–12, 2022.