Accelerating GNSS Software Receivers

Carles Fernández–Prades, Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), Spain.
Javier Arribas, Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), Spain.
Pau Closas, Centre Tecnològic de Telecomunicacions de Catalunya (CTTC), Spain.

BIOGRAPHY

Dr. Carles Fernández–Prades holds the position of Senior Researcher and served as Head of the Communications Systems Division (2013-2016) and the Communication Susbystems Area (2006-2013) at the Centre Tecnològic de Telecomunicacions de Catalunya (CTTC). He received a PhD degree in Electrical Engineering from Universitat Politècnica de Catalunya (UPC) in 2006. His primary areas of interest include statistical and multi-sensor signal processing, estimation and detection theory, and Bayesian filtering, with applications related to communication systems, GNSS and software-defined radio technology.

Dr. Javier Arribas holds the position of Senior Researcher at the Centre Tecnològic de Telecomunicacions de Catalunya (CTTC). He received the BSc and MSc degree in Telecommunication Engineering in 2002 and 2004, respectively, at La Salle University in Barcelona, Spain. His primary areas of interest include statistical signal processing, GNSS synchronization, detection and estimation theory, software defined receivers, FPGA prototyping and the design of RF front-ends.

Dr. Pau Closas is Head of the Statistical Inference for Communications and Positioning Department and is a Senior Researcher at the Centre Tecnològic de Telecomunicacions de Catalunya. He received the MSc and PhD degrees in Electrical Engineering from Universitat Politècnica de Catalunya in 2003 and 2009, respectively. He also holds a MSc degree in Advanced Mathematics and Mathematical Engineering from UPC since 2014. During 2008 he was a Research Visitor at the Stony Brook University, New York, USA. His primary areas of interest include statistical and array signal processing, estimation and detection theory, Bayesian filtering, robustness analysis, and game theory, with applications to positioning systems and wireless communications.

ABSTRACT

This paper addresses both the efficiency and the portability of a computer program in charge of the baseband signal processing of a GNSS receiver. Efficiency, in this context, refers to optimizing the speed and memory requirements of the software receiver. Specifically, the interest is focused on how fast the software receiver can process the incoming stream of raw signal samples and, in particular, if signal processing up to the position fix can be executed in real-time (and how many channels the host computer executing the receiver application can sustain in parallel). This is achieved by applying the concept of parallelization at different abstraction levels. The paper describes strategies based on task, data and instruction-level parallelism, as well as actual implementations released under an open source license and the results obtained with different commercially available computing platforms. At the same time, the proposed solution also addresses portability, understood as the usability of the same software in different computing environments.

1. INTRODUCTION

Back in 2006, Gregory W. Heckler published an open source library implementing SIMD-based correlators for GNSS software receivers [1]. That library included arithmetic functions that operate on 16 bit integers, providing MMX and SSE2 versions of each function, as well as an MMX-enabled fixed point, radix-2, FFT. The code was then integrated into a software-defined GPS L1 C/A receiver (still available online[1]), which became, to the authors knowledge, the first available open source software-defined GPS receiver that attained real-time processing in midrange computers.

In one decade, processors' clocking speed has hardly doubled. A typical desktop bought in 2006 could be shipped with a Pentium IV processor at 2.0 GHz, with the 64-bit x64_86

[1] See `https://github.com/gps-sdr/gps-sdr`

architecture being introduced in the market. In 2016, Intel is planning to release their Broadwell-E processor series, with a clock speed up to 3.50 GHz. Differences are more obvious in the number of cores and the memory bandwidth: while in 2006 dual-core processor technology was being introduced in the market, with memory bandwidths on the order of 2 GB/s, Broadwell-E processors can house up to 20 cores, with memory bandwidths on the order of 100 GB/s. The requirements for a GNSS software receiver have also evolved in such period: from targeting GPS L1 C/A signals transmitted by 24 satellites, GNSS receiver technology is moving into a scenario with more than 100 satellites, 10 different GNSS open signal waveforms for civilian usage, belonging to 4 different systems and broadcast at 4 different frequency bands. The new signals are mostly based on binary offset carrier (BOC) modulations, which require more bandwidth and more processing complexity on the receiver, and the natural target is a multi-constellation, multi-band GNSS receiver operating in real-time. It is then of the utmost importance to exploit the underlying parallelisms in the processing platform executing the software receiver in order to meet real-time requirements.

After describing the technical foundations on how the typical mathematical operations involved in GNSS baseband processing can be accelerated by honoring certain software design patterns and exploiting the underlying hardware architecture of the processing platform, this paper presents a practical, reusable, and expandable implementation that meets another important feature in nowadays computing ecosystem: portability. Solutions are tested in disparate, commercially available platforms, providing performance measurements both as isolated operations and in the context of a GNSS software receiver, and releasing all the source code under an open source license. As a practical outcome of the presented work, the paper presents the first free and open source software defined receiver that is able to operate in real-time in ARM-based, commercially available processing platforms, as well as an independent software library that can be used by other software applications.

2. PARALLELIZATION STRATEGIES

2.1. Task parallelism

A fundamental model of architectural parallelism is found in shared-memory parallel computers, which can work on several tasks at once, simply by parceling them out to the different processors, by executing multiple instruction streams in an interleaved way in a single processor (an approach known as simultaneous multithreading, or SMT), or by a combination of both strategies. SMT platforms, multicore machines, and shared-memory parallel computers all provide system support for the execution of multiple independent instruction streams, or *threads*. This approach is referred to as *task parallelism*, and it is well supported by the main programming languages, compilers and operating systems [2]. To make this potential performance gain effective, the software running on the platform must be written in such a way that it can spread its workload across multiple execution cores. Applications and operating systems that are written to support this feature are referred to as *multi-threaded*. When programmed with the appropriate design, execution can be accelerated almost linearly with the number of processing cores.

2.2. Data parallelism

In order to cope with the high computational load of GNSS baseband processing, software-defined receivers can resort to another form of parallelization: instructions that can be applied to multiple data elements in parallel, thus exploiting *data parallelism*. This computer architecture is known as *Single Instruction Multiple Data* (SIMD). Intel introduced the first instance of SIMD extensions to the 32-bit x86 architecture, called MMX, in 1997. MMX added eight 64-bit registers to the x86 instruction set, and a plethora of SIMD operations to operate on the data in those registers. Later SIMD extensions, named SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2, added eight 128-bit registers to the x86 instruction set. Additionally, SSE operations included SIMD floating-point operations, and expanded the type of integer operations available to the programmer. The other important processor manufacturer, AMD, quickly followed this technology and thus it is present in the vast majority of modern personal computers. Starting from Intel's Sandy Bridge processors (2011), a new instruction set called Advanced Vector Extensions (AVX) was incorporated which has the capacity to further accelerate the computation of the vector operations. AVX provides new features, 256-bit registers, new instructions and a new coding scheme. AVX-2 was introduced with Haswell processors (2013), expanding most integer commands to 256 bits, and Intel's Xeon Phi processors (expected in 2016) will support AVX-512, providing 512-bit extensions to AVX. SIMD technology is also present in embedded systems: NEON technology is a 128-bit SIMD architecture extension for the ARM Cortex-A series processors, providing 32 registers, 64-bits wide (dual view as 16 registers, 128-bits wide).

2.3. Instruction-level parallelism

Computer processors are composed of a number of functional units that may be able to operate simultaneously. As a result, a computer might be able to fetch a datum from memory, multiply two floating-point numbers, and evaluate a branch condition all at the same time. This is often referred to as *instruction-level parallelism*. A processor that supports this is said to have a *superscalar* architecture, and nowadays it is a common feature in general-purpose microprocessors. Modern compilers put considerable effort into finding a suitable ordering of operations that keeps many functional units and

paths to memory busy with useful work. Unfortunately, several studies showed that typical applications are not likely to contain more than three or four different instructions that can be fed to the computer at a time in this way, limiting the reach of this approach [3]. Even so, techniques such as manual loop unrolling can still accelerate execution by a factor of two [1].

3. COMPUTING ECOSYSTEM

In 2016, desktop computers are not the dominant form factor anymore. Laptops, gaming consoles, mini PCs, tablets and smartphones has pushed into the market other sort of processors with low power consumption figures and specific features for multimedia content handling. This is the case of the embedded reduced instruction set computing (RISC) microprocessor architecture or, most commonly referred to by the name of the industry's leading provider, ARM processors. Starting from Cortex-A series, they support SIMD through NEON technology, a 128-bit architecture extension consisting of a set of registers and operands designed to accelerate image processing tasks such as 2D/3D graphics manipulation and video encoding and decoding. In 2016, ARM-based multi-core systems-on-chip are already available in commercial devices such as smartphones and tablets. As a well-known and accessible example, Raspberry Pi 3 ships with a quad-core 64-bit ARM Cortex A53 running at 1.2 GHz. This paper presents an open source library that implements NEON versions of the most demanding mathematical operations in a typical GNSS software receiver, showing accelerations surpassing a factor of ten when executed in actual ARM-based systems, with the overall result of enabling real-time execution in such devices.

Another relevant example is found in Graphic Processing Units, or GPUs, which are specifically designed to accelerate the creation of images in a frame buffer intended for output to a display. GPUs are commonplace in embedded systems, mobile phones, personal computers, workstations, high-end video cards and game consoles. Such processor architecture follows another parallel programming model, called *Single Instruction, Multiple Threads* (SIMT). While in SIMD elements of short vectors are processed in parallel, and in SMT instructions of several threads are run in parallel, SIMT is a hybrid between vector processing and hardware threading. Currently, OpenCL is the most popular open GPU computing language that supports devices from several manufacturers, while CUDA is the dominant proprietary framework specific for NVIDIA GPUs. GPUs are massive parallel computing platforms that can hold hundreds or even thousands of stream processors. However, each of them runs slower than a CPU core does and, even though they are Turing complete, they miss some essential features such as virtual memory managers or hardware interrupt controllers, and their instruction sets are optimized mostly for image processing using floating point data. The use of GPUs to accelerate a software-defined GNSS receiver is not a straightforward process, and it requires solving several trade-offs to split the signal processing chain between the host CPU and the auxiliary GPU (or GPUs) in order to use the best complementing features of both architectures [4]. This paper describes a GPU-based implementation of GNSS multiple correlators that can be seamlessly integrated into a GNSS software-defined receiver being executed in a general-purpose processor, sharing the processing work with the CPU and thus alleviating the overall load.

This rich ecosystem of computing platforms (*i.e.*, several generations of Intel and AMD processors, implementing different SIMD instruction sets; ARM processors with NEON extensions, all in 32 and 64-bit architectures; and the possible presence of GPUs) raises the need to address portability as a key feature for a real impact in an open source context.

The baseband processing acceleration strategies for GNSS software-defined receivers described on this paper are put in practice in GNSS-SDR [5], an open source software receiver available online[2], that can be executed in Intel, AMD and ARM processors (32 and 64 bits), thus covering the vast majority of today's computers.

4. TASK PARALLELIZATION

Task parallelization focuses on distributing execution processes (threads) across different parallel computing nodes (processors), each executing a different thread (or process) on the same or different data. Spreading processing tasks along different threads must be carefully designed in order to avoid bottlenecks (either in the processing or in memory access) that can block the whole processing chain and prevent it from attaining real-time operation. This section provides an overview of the task scheduling strategy implemented in GNSS-SDR and a description of the most computationally demanding operations in a GNSS receiver.

4.1. Multi-threading in GNSS software receivers

GNSS-SDR uses a "thread-per-block" scheduler, which means that each instantiated processing block runs in its own thread. This architecture scales very well to multicore processor architectures. The implementation is provided by GNU Radio [6], whose flow graph computations can be jointly modeled as a Kahn process [7, 8]. A Kahn process describes a model of computation where processes are connected by communication channels to form a network. Processes produce data elements or tokens and send them along a communication channel where they are consumed by the waiting destination process. Communication channels are the only method

[2]See `https://github.com/gnss-sdr/gnss-sdr`

processes may use to exchange information. Kahn requires the execution of a process to be suspended when it attempts to get data from an empty input channel. A process may not, for example, test an input for the presence or absence of data. At any given point, a process can be either enabled or blocked waiting for data on only one of its input channels: it cannot wait for data from more than one channel. Systems that obey Kahn's mathematical model are determinate: the history of tokens produced on the communication channels does not depend on the execution order [7]. With a proper scheduling policy, it is possible to implement software defined radio process networks holding two key properties:

- Non-termination: understood as an infinite running flow graph process without deadlocks situations, and
- Strictly bounded: the number of data elements buffered on the communication channels remains bounded for all possible execution orders.

An analysis of such process networks scheduling was provided in [9]. By adopting GNU Radio's signal processing framework, GNSS-SDR bases its software architecture in a well-established design and extensively proven implementation. Section 6.1 provides details on how this concept is applied in the context of a GNSS software-defined receiver.

4.1.1. GPU Offloading

GPU-accelerated computing consists in the use of a graphics processing unit (GPU) together with a CPU to accelerate the execution of a software application, by offloading computation-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. The key idea is to utilize the computation power of both CPU cores and GPU execution units in tandem for better utilization of available computing power. Examples of GPU offloading in the context of GNSS receivers have been extensively reported in literature (see, for instance, [10, 11, 12]).

4.1.2. FPGA Offloading

The commercial availability of system-on-chip (SoC) devices which integrate the software programmability of an ARM-based processor with the hardware programmability of an FPGA (*e.g.*, Xilinx's Zynq-7000 family [13]), allows for systems in which the most computationally demanding operations of the GNSS receiver are executed in the programmable logic, whereas the rest of the software receiver is executed in the processing system. The implementation of FPGA-based accelerators and its communication with processes executed in the ARM processor are out of the scope of this paper.

4.2. Key operations and data types

In order to describe the most computationally demanding operations in the receiver chain, let us assume a generic GNSS complex baseband signal of the form

$$s_T(t) = \sqrt{P_T} \sum_{u=-\infty}^{\infty} d(u) p(t - uT_{b_I}) \,, \tag{1}$$

where

$$p(t) = \sum_{k=0}^{N_c-1} q(t - kT_{PRN}) \tag{2}$$

and

$$q(t) = \sum_{l=0}^{L_c-1} c_i(l) g_T(t - lT_c) \,, \tag{3}$$

being P_T the transmitting power, $d(u) \in \{-1, 1\}$ the navigation message data symbols, T_b the bit period, N_c the number of repetitions of a full codeword that spans a bit period, $T_{PRN} = \frac{T_b}{N_c}$ the codeword period, $c_i(l) \in \{-1, 1\}$ a chip of a spreading codeword i of length L_c chips, $g_T(t)$ the transmitting chip pulse shape, which is considered energy-normalized for notation clarity, and $T_c = \frac{T_b}{N_c L_c}$ is the chip period.

The analytic representation of a signal received from a generic GNSS satellite i can be generically expressed as

$$r_i(t) = \alpha_i(t) s_{i,T}\left(t - \tau_i(t)\right) e^{-j2\pi f_{d_i}(t)} e^{j2\pi f_c t} + w(t) \,, \tag{4}$$

where $\alpha_i(t)$ is the amplitude, $s_{i,T}(t)$ is the complex baseband transmitted signal, $\tau_i(t)$ is the time–varying delay, $f_{d_i}(t) = f_c \tau_i(t)$ is the Doppler shift, f_c is the carrier frequency, and $w(t)$ is a noise term.

Assuming $w(t)$ as additive white Gaussian noise, at least in the band of interest, it is well known that the optimum receiver is the code matched filter (often referred to as *correlator*), expressed as

$$\begin{aligned} h_{MF_i}(t_k; \hat{\tau}_i, \hat{f}_{d_i}, \hat{\phi}_i) = \sum_{l=0}^{L_c-1} c_i^*(l) g_R^*(-t_k - lT_c + \hat{\tau}_i + L_c T_c) \cdot \\ \cdot e^{-j\hat{\phi}_i} e^{-j2\pi \hat{f}_{d_i} t_k} = \\ = q_R^*(-t_k + \hat{\tau}_i + L_c T_c) e^{-j\hat{\phi}_i} e^{-j2\pi \hat{f}_{d_i} t_k} \,, \end{aligned} \tag{5}$$

where $c_i(l) \in \{-1, +1\}$ is the l-th chip of a spreading codeword (known as pseudorandom sequence) of length L_c, $g_R(t)$ is the receiving chip pulse shape, T_c is the chip period, and $\hat{\tau}_i, \hat{f}_{d_i}, \hat{\phi}_i$ are local estimates of the time-delay, Doppler-shift and carrier phase of the received signal, respectively. The code matched filter output can be written as a convolution of the form

$$\begin{aligned} y_i(t_k; \hat{\tau}_{i_{k-1}}, \hat{f}_{d_{i_{k-1}}}, \hat{\phi}_{i_{k-1}}) = \\ = r_i(t_k; \tau_{i_k}, f_{d_{i_k}}, \phi_{i_k}) * h_{MF_i}(t_k; \hat{\tau}_{i_{k-1}}, \hat{f}_{d_{i_{k-1}}}, \hat{\phi}_{i_{k-1}}) \,. \end{aligned} \tag{6}$$

Notice that, in the matched filter, we have substituted the estimates $\hat{\tau}_{i_k}$, $\hat{f}_{d_{i_k}}$ and $\hat{\phi}_{i_k}$ for trial values obtained from previous (in time) estimates of these parameters, which we have

defined as $\hat{\tau}_{i_{k-1}}$, $\hat{f}_{d_{i_{k-1}}}$ and $\hat{\phi}_{i_{k-1}}$, respectively. This is the usual procedure in GNSS receivers, since the estimates are not really available, but are to be estimated after correlation. Since the correlators perform the accumulation of the sampled signal during a period T_{int} and then release an output, we can write the discrete version of the signal as:

$$y_{i_k} = \frac{|a_{i_k}|}{2} K \frac{\sin(\pi \Delta f_{d_{i_k}} T_{int})}{\pi \Delta f_{d_{i_k}} T_{int}} \cdot d_i \left([k]_{\frac{T_b}{T_{int}}} \right) \cdot \\ \cdot R_{\tilde{p}q}(\Delta \tau_{i_k}) \cdot e^{-j(\pi \Delta f_{d_{i_k}} T_{int} + \Delta \phi_{i_k})} + \tilde{w}_{i_k} \quad (7)$$

where we defined a_{i_k} as the receiving signal complex amplitude of the GNSS satellite i at time k, $\Delta f_{i_k} = f_{d_{i_k}} - \hat{f}_{d_{i_{k-1}}}$, $\Delta \phi_{i_k} = \phi_{i_k} - \hat{\phi}_{i_{k-1}}$ and $\Delta \tau_{i_k} = \tau_{i_k} - \hat{\tau}_{i_{k-1}}$ (that is, the estimation errors of Doppler shift, carrier phase and time delay, respectively), T_{int} is the integration time, $R_{\tilde{p}q}(t) = \int_{-\infty}^{+\infty} \tilde{p}^*(\psi) q(t-\psi) d\psi$ is the cross-correlation function between the received, possibly filtered pulse train modulated by the PRN sequence, and a local replica $q(t)$ defined as in (3), $\tilde{w}_{i_k}$ as remnant noise, and $[k]_{\frac{T_b}{T_{int}}}$ means the integer part of $\frac{kT_{int}}{T_b}$. From now on, we will consider $K = \frac{T_{int}}{T_s}$ as the integer number of samples collected in an accumulation. This number will not be an integer in receiver configurations having a sample rate incommensurable with the chip rate, and thus some integration blocks will accumulate $K+1$ samples instead of K.

The output of the correlator centered at the estimation of the time delay error in the previous time step, known as *Prompt* correlator, can be the written as:

$$P_{i_k} = y_{i_k} \left(\Delta \hat{\tau}_{i_{k-1}}, \Delta \hat{f}_{d_{i_{k-1}}}, \Delta \hat{\phi}_{i_{k-1}} \right) . \quad (8)$$

The other correlators are shifted in time with respect to the Prompt, usually in a symmetric arrangement. The advanced and delayed replicas are known as *Early* and *Late* correlator outputs, and can be written as

$$E_{i_k} = y_{i_k} \left(\Delta \hat{\tau}_{i_{k-1}} + \epsilon, \Delta \hat{f}_{d_{i_{k-1}}}, \Delta \hat{\phi}_{i_{k-1}} \right) \quad (9)$$

and

$$L_{i_k} = y_{i_k} \left(\Delta \hat{\tau}_{i_{k-1}} - \epsilon, \Delta \hat{f}_{d_{i_{k-1}}}, \Delta \hat{\phi}_{i_{k-1}} \right) . \quad (10)$$

Very Early and *Very Late* outputs can be defined likewise with $\epsilon' > \epsilon$. From those correlator outputs, some error functions (known in this context as *discriminators*) can be defined [14]. For instance, the normalized non-coherent code discriminator

$$\Delta \hat{\tau}_{i_k} = \frac{|E_{i_k}|^2 - |L_{i_k}|^2}{\sqrt{\frac{1}{\mathcal{U}} \sum_{u=0}^{\mathcal{U}-1} \left| \Re\{P_{i_{k-u}}\} \right|^2 + \frac{1}{\mathcal{U}} \sum_{u=0}^{\mathcal{U}-1} \left| \Im\{P_{i_{k-u}}\} \right|^2}} , \quad (11)$$

the coherent carrier phase four-quadrant arctangent discriminator

$$\Delta \hat{\phi}_{i_k} = \text{atan2} \left(\frac{\Im\{P_{i_k}\}}{\Re\{P_{i_k}\}} \right) , \quad (12)$$

and the four-quadrant arctangent FLL discriminator as a measure of the frequency error

$$\Delta \hat{f}_{i_k} = \frac{1}{2\pi (t_k - t_{k-1})} \cdot \\ \cdot \text{atan2} \left(\frac{\Re\{P_{i_{k-1}}\}\Re\{P_{i_k}\} + \Im\{P_{i_{k-1}}\}\Im\{P_{i_k}\}}{\Re\{P_{i_{k-1}}\}\Im\{P_{i_k}\} - \Re\{P_{i_k}\}\Im\{P_{i_{k-1}}\}} \right) , \quad (13)$$

where $\Re\{\cdot\}$ and $\Im\{\cdot\}$ stand for the real and imaginary components, respectively.

The new estimations $\Delta \hat{\tau}_{i_k}$, $\Delta \hat{\phi}_{i_k}$ and $\Delta \hat{f}_{i_k}$ are then low-pass filtered and reinjected back into the matched filter of Equation (5), thus closing the tracking loops. Such an architecture is represented in Figure 1. Delay estimation is used to compute pseudorange, while phase and frequency estimators are used to demodulate and decode the navigation message (that is, the sequence of bits denoted as $d(u)$ in Equation (1)) and to obtain phase range and phase range rate observables. This is all the information required to compute a position fix.

At this level of abstraction, parallelization of operations to the incoming signal is performed at a targeted satellite signal basis. Explicitly, the stream of digital signal samples at the output of the analog-to-digital converter of a radio-frequency front-end, assuming N_s in-view GNSS satellites, can be written as

$$x_q[n] = \sum_{i=0}^{N_s-1} \tilde{\alpha}_i(t_n) \tilde{s}_{i,T}(t_n - \tau_{i_n}) e^{-j2\pi f_{d_{i_n}} t_n} e^{j2\pi f_{\text{IF}} t_n} + \tilde{w}(t_n) \quad (14)$$

where $t_n = nT_s$, $T_s \leq 2T_c$ is the sampling period, n is the temporal index, f_{IF} is the intermediate frequency, and $\tilde{\cdot}$ denotes a filtered, probably distorted version of the original signal. The correlations described in Equation (6) and both the delay and phase locked loops are replicated for each received satellite signal (denoted in the equation with the index i), with a minimum of four of them locked-in at the same time in order to be able to compute a position fix.

The index q in Equation (14) denotes that the value of each sample $x_q[n]$ is actually quantized and represented with q bits. Most commercial RF integrated circuits for GNSS receivers deliver samples with $q = 2$ or $q = 3$ bits. However, this is not a format that a computer is ready to manipulate internally. Even though operations involving values expressed with 3 bits can be defined by software, neither the compiler nor the underlying hardware are optimized to do so. Processor architectures are designed to work more efficiently when manipulating values with specific bit lengths (that is, 8, 16, 32

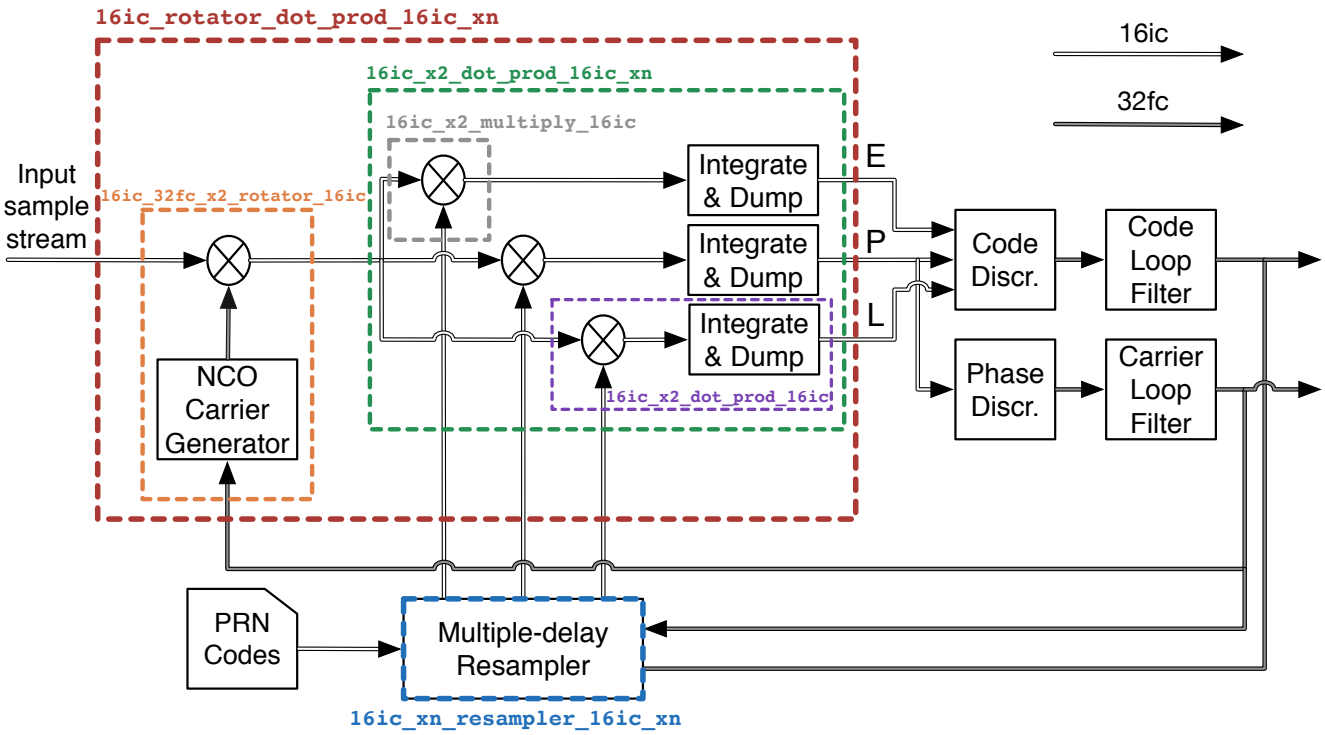

Fig. 1. Diagram of typical code and carrier tracking loops in a GNSS receiver. Colored dotted-line boxes show functions that have been implemented in SIMD technology. In this example, lanes with label "16ic" are data streams whose items are complex numbers with real and imaginary components represented with 16-bit integers, whereas label "32fc" indicates lanes whose items are complex numbers with real and imaginary components in 32-bit floating point representation.

or 64 bits per item), and interpreted either as integers (signed or unsigned) or floating-point values. Some of those specific formats for data items are summarized in Table 1. A conversion is then required from the sample bit length delivered by the analog-to-digital converter at the output of the front-end to the bit length and format of the data items feeding the software-defined receiver. Section 7 presents results for operations (shown in Figure 1) on data types labelled as "16ic" and "32fc" in Table 1.

5. DATA PARALLELIZATION

At a lower level of abstraction, some operations on incoming data can be further parallelized by applying the same operation on different data samples at a time (*i.e.*, parallelizing in the temporal index n). This is the approach of SIMD processing, which has been embodied in different technologies described below.

5.1. SSE technology

The family of Streaming SIMD Extensions (SSE) instruction sets is now present in all Intel and AMD processors of today's computers. In this technology, the same set of instructions is executed in parallel to different sets of data. This reduces the amount of hardware control logic needed by N times for the same amount of calculations, where N is the width of the SIMD unit. In case of SSE, registers are 128-bits wide, so each one can hold two complex floating point samples (denoted as "32fc" in Table 1), or four complex short integers (denoted as "16ic" in Table 1). Operations are then applied to those registers, thus executing the same instruction to multiple samples at a time, and saving clock cycles. Hence, SIMD operations can only be applied to certain predefined processing patterns. In addition, it is important to take into account that Intel's and AMD's processors will transfer data to and from memory into registers faster if the data is aligned to 16-byte boundaries. While the compiler will take care of this alignment when using the basic 128-bit type, this means that data has to be stored in sets of four 32-bit floating point values in memory for optimal performance. If data is not stored in this kind of fashion then more costly unaligned scalar memory moves are needed, instead of packaged 128-bit aligned moves. Effective SSE will minimize the number of data movements between the memory subsystem and the CPU registers. The data should be loaded into SSE registers only once, and then the results moved back into memory only once when they are no longer needed in that particular code

Type name in VOLK	Definition	Sample stream
"8i"	Signed integer, 8-bit two's complement number ranging from -128 to 127. C type name: `int8_t`	$[S_0], [S_1], [S_2], ...$
"8u"	Unsigned integer, 8 bits ranging from 0 to 255. C type name: `unsigned char`	$[S_0], [S_1], [S_2], ...$
"8ic"	Complex samples, with real and imaginary parts of type `int8_t` C type name: `lv_8sc_t` (*)	$[S_0^I + jS_0^Q], [S_1^I + jS_1^Q], ...$
"16i"	Signed integer, 16-bit two's complement number ranging from -32768 to 32767 C type name: `int16_t`	$[S_0], [S_1], [S_2], ...$
"16u"	Unsigned integer, 16 bits ranging from 0 to 65535. C++ type name: `uint16_t`	$[S_0], [S_1], [S_2], ...$
"16ic"	Complex samples, with real and imaginary parts of type `int16_t` C type name: `lv_16sc_t` (*)	$[S_0^I + jS_0^Q], [S_1^I + jS_1^Q], ...$
"32u"	Unsigned integer, 32 bits ranging from 0 to 4294967295. C type name: `uint32_t`	$[S_0], [S_1], [S_2], ...$
"32f"	Signed numbers with fractional parts, can represent values ranging from $\approx 3.4 \times 10^{-38}$ to 3.4×10^{38} with a precision of 7 digits (32 bits). C type name: `float`	$[S_0], [S_1], [S_2], ...$
"32fc"	Complex samples, with real and imaginary parts of type `float` C++ type name: `lv_32fc_t` (*)	$[S_0^I + jS_0^Q], [S_1^I + jS_1^Q], ...$
"64u"	Unsigned integer, 64 bits ranging from 0 to $2^{64} - 1$. C type name: `uint64_t`	$[S_0], [S_1], [S_2], ...$
"64f"	Signed numbers with fractional parts, can represent values ranging from $\approx 1.7 \times 10^{-308}$ to 1.7×10^{308} with a precision of 15 digits (64 bits). C type name: `double`	$[S_0], [S_1], [S_2], ...$

Table 1. Data type names used in the VOLK library, which also provides the C programming language type name definitions marked with an asterisk (*).

block. Listing 1 provides a pseudocode example of SIMD programming.

5.2. AVX technology

Intel's extension to the SSE family is the Advanced Vector Extension (AVX), which extends the 128-bit SSE register into 256-bit AVX register that consist of two 128-bit lanes. An AVX lane is an extension of SSE4.2 functionality, with each register holding eight samples of complex type 16-bit integer or four samples of complex type 32-bit float. AVX operates most efficiently when the same operations are performed on both lanes. On the contrary, cross-lane operations are limited and expensive, and not all bit shuffling combinations are allowed. This leads to higher shuffle-overhead since many operations now require both cross-lane and intra-lane shuffling [15], thus expending extra clock cycles. The same applies to AVX-2 (which adds integer operations to the AVX instruction set), and most recent AVX-512, which introduces 512-bit wide registers.

5.3. NEON technology

SIMD technology is also present in ARM processors through the NEON instruction set. The NEON instructions support 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers, as well as 32-bit single-precision floating point elements. NEON technology includes support for unaligned data accesses and easy loading of interleaved data, so there is no need to account for that, on the contrary of SSEx and AXV. A draw back is that the NEON floating point pipeline is not entirely IEEE-754 compliant. This is a problem for blocks when processing a large large number of floating point items, since the different results will accumulate along samples and makes NEON and SSE results not comparable. Countermeasures should be taken where applicable.

6. IMPLEMENTATION

6.1. A multi-threaded GNSS receiver

Software defined receivers can be represented as flow graph of nodes. Each node represents a signal processing block, whereas links between nodes represents a flow of data. The concept of a flow graph can be viewed as an acyclic directional graph with one or more source blocks (to insert samples into the flow graph), one or more sink blocks (to terminate or export samples from the flow graph), and any signal processing blocks in between. The diagram of a processing block (that is, of a given node in the flow graph), as implemented by the GNU Radio framework, is shown in Figure 2. Each block can have an arbitrary number of input and output *ports* for data and for asynchronous message passing with other blocks in the flow graph. In all software applications based on the GNU Radio framework, the underlying process scheduler passes items (i.e., units of data) from sources to sinks. For each block, the number of items it can process in a single iteration is dependent on how much space it has in its output buffer(s) and how many items are available on the input buffer(s). The larger that number is, the better in terms of efficiency (since the majority of the processing time is taken up with processing samples), but also the larger the latency that will be introduced by that block. On the contrary, the smaller the number of items per iteration, the larger the overhead that will be introduced by the scheduler.

Thus, there are some constraints and requirements in terms of number of available items in the input buffers and in available space in the output buffer in order to make all the processing chain efficient. In GNU Radio, each block has a

Algorithm 1 Simplified pseudocode for each block's thread in GNU Radio.

1: Set thread's processor affinity and thread priority.
2: Handle queued messages.
3: Compute items available on input buffer(s) as the difference between write and read pointers for all inputs.
4: Compute space on output buffer(s) as the difference between write pointers to the first read pointer.
5: **if** all requirements are fulfilled **then**
6: Get all pointers to input and output buffers.
7: Execute the actual signal processing (call `work()`).
8: **else**
9: Try again.
10: **end if**
11: Notify neighbors (tell previous and next blocks that there is input data and/or output buffer space).
12: Propagate to upstream and downstream blocks that the iteration has finalized.
13: Wait for data/space or a new message to handle.

runtime scheduler that dynamically performs all those computations, using algorithms that attempt to optimize throughput, implementing a process network scheduling that fulfills the requirements described in [9]. Each processing block executes in its own thread, which runs the procedure sketched in Algorithm 1. A detailed description of the GNU Radio internal scheduler implementation (memory management, requirement computations, and other related algorithms and parameters) can be found in [16], and of course in GNU Radio source code[3].

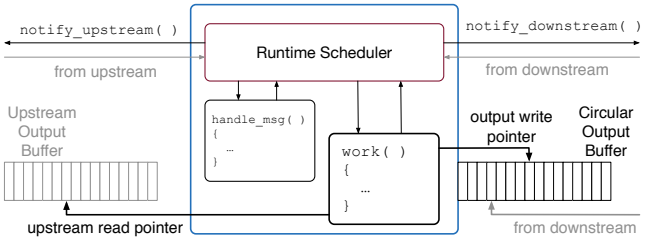

Fig. 2. Diagram of a signal processing block, as implemented by GNU Radio. Each block has a completely independent scheduler running in its own execution thread and a messaging system for communication with other upstream and downstream blocks. The actual signal processing is performed in the `work()` method. Figure adapted from [17].

Under this scheme, software-defined signal processing blocks read the available samples in their input memory buffer(s), process them as fast as they can, and place the result in the corresponding output memory buffer(s), each of them being executed in its own, independent thread. This strategy results in a software receiver that always attempts to process signal at the maximum processing capacity, since each block in the flow graph runs as fast as the processor, data flow and buffer space allows, regardless of its input data rate. Achieving real-time is *only* a matter of executing the receivers full processing chain in a processing system powerful enough to sustain the required processing load, but it does not prevent from executing exactly the same process at a slower pace, for example, by reading samples from a file in a less powerful platform.

Figure 3 shows the flow graph diagram used in GNSS-SDR. There is a signal source block (either a file or a radio-frequency front-end) writing samples in a memory buffer at a given sampling rate; some signal conditioning (possible data type adaptation, filtering, frequency downshifting to baseband, and resampling); a set of parallel channels, each one reading form the same upstream buffer and targeted to a different satellite; a block in charge of the formation of observables collecting the output of each satellite channel after the despreading (and thus in a much slower rate); and a signal sink, responsible for computing the position-velocity-time solution from the obtained observables and providing outputs in standard formats (such as KML, GeoJSON, RINEX, RTCM and NMEA).

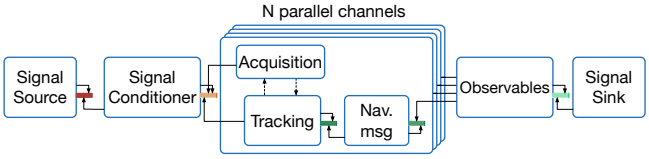

Fig. 3. Simplified GNSS-SDR flow graph diagram. Each blue box is a signal processing block sketched in Figure 2. Here, only the data stream layer is represented, where different data rates are indicated with different colors for the memory buffers.

The flow graph in Figure 3 can be expanded to accommodate more GNSS signal definitions in the same band (for instance, a GPS L1 C/A and Galileo E1b receiver), and to accommodate more bands (thus defining a multi-band, multi-system GNSS receiver). In all cases, each of the processing blocks will be executing its own thread, defining a multi-threaded GNSS receiver that efficiently exploits task parallelization.

6.2. Single Instruction, Multiple Data (SIMD)

The implementation of data parallelization techniques presented in this paper extends the Vector-Optimized Library of Kernels (VOLK[4], see [18, 19, 20]), an open source software

[3] Available at `https://github.com/gnuradio/gnuradio`

[4] Available at `https://github.com/gnuradio/volk`

library that provides an abstraction of optimized mathematical routines targeting several SIMD processors. For each targeted mathematical operation (referred to as *kernels* in this context), VOLK provides a "generic" (i.e. written in plain C language) implementation that can run in virtually every modern processor, plus a number of versions for different SIMD technologies. For each architecture or platform that a developer wishes to vectorize for (e.g. SSE, AVX, NEON, etc.), a new implementation of a given function can be added to VOLK. The specific host computer devoted to run the software receiver is then benchmarked by executing all the implementations that it is able to do. At runtime, when such functions are called by a software application, VOLK selects the fastest implementation available for the given processor and memory alignment, thus addressing efficiency and portability at the same time. Each specific kernel implementation for a given platform, SIMD technology version or run-time condition is known as *proto-kernel*.

As an additional key feature, VOLK provides `volk_modtool`, an application that generates an *empty* copy of VOLK's software structure (basically all the source code tree, with the benchmarking system, architecture and machine abstractions, automatic selection of best implementation at runtime, building scripts, etc., but without including any kernel implementation) ready to be filled with other custom kernels that, by its specificity, are not included in the library. Benefiting from such modular design, and VOLK's open source license, the authors of this paper developed a complementary new module, so-called VOLK_GNSSSDR, which extends the original VOLK with operations that are useful in the context of a GNSS software receiver, such as joint Doppler removal and correlation of the incoming signals with multiple delayed local replicas of the PRN code (as shown in Figure 1). Although this new library is used by GNSS-SDR, it can also be used independently for other purposes.

A pseudocode example of a proto-kernel is shown in Listing 1. The approach consists in processing as many input data items as possible by packing such items into SIMD registers, and computing the desired result using vector operands, either using in-line assembly or compiler intrinsics. The "tail" items that do not fit in those packs are then processed separately, in a plain C implementation.

```
static inline void
volk_gnsssdr_16ic_x2_multiply_16ic_neon(
    lv_16sc_t* out, const lv_16sc_t* in_a,
    const lv_16sc_t* in_b, unsigned int num_points)
{
  const unsigned int neon_iters = num_points / 4;
  /* neon_iters only integer part of division!  */
  unsigned int n;
  /* Declare other local variables              */
  for(n = 0; n < neon_iters; ++n)
    {
      /* NEON implementation                     */
      /*    1) Load 4 items from in_a and in_b   */
      /*       in NEON 128-wide registers        */
      /*    2) Multiply four complex numbers     */
      /*       using NEON vector operands        */
      /*    3) Store results in out pointer      */
    }
  for (n = neon_iters * 4; n < num_points; ++n)
    {
      /* Plain C implementation for tail items   */
      /* that do not fit in 4-item packets       */
      out[n] = in_a[n] * in_b[n];
    }
}
```

Listing 1. Example of data parallelization implementation: C code structure for vector element-wise multiplication (16-bit integer complex data type) in NEON technology. Each data item is 32-bit wide, so 4 items fit in a 128-bit register which can then be operated at the same time, thus saving clock cycles.

6.3. Single Instruction, Multiple Threads (SIMT)

In the SIMT execution model, the SIMD model is combined with multi-threading. In a SIMT machine, there is a set of processors, each of them enabled to execute N parallel threads performing the same instruction. The key difference between SIMD and SIMT is that in the latter it is possible to combine a single instruction with *multiple registers*, *multiple addresses* and *multiple flow paths*. This feature enables the parallel processing of large vectors of data, and it is specially well-suited to speed up the real-time high-resolution 3D graphics computing-intensive tasks, such as texture processing. This execution model establishes a hierarchy of execution units, such as *blocks* (set of threads) and *grids* (set of blocks). In a GPU kernel call, the programmer must select the number of threads per block and the number of blocks per grid, and this will impact on the amount of computational resources allocated during the call. For more detailed information the reader is referred to [21]. In a GPU-accelerated GNSS receiver, each of the receiver channels must share the GPU resources, therefore, there is a trade-off in the resources allocation for each channel and the maximum number of available channels. If the GPU resource occupancy reaches 100%, the GPU kernel calls are queued for later execution, so the integration must be carefully designed in order to avoid blocking the whole flow graph execution.

For a practical implementation, we opted for the NVIDIA GPU computing platform and the CUDA programming model [22]. A CUDA kernel was implemented integrating the operations of the VOLK_GNSSSDR kernels "32fc_rotator_dot_prod_32fc_xn" and "32fc_xn_resampler_32fc_xn", as defined in Figure 1, and

embedded into GNSS-SDR as an option for signal tracking implementation.

7. RESULTS

The presented VOLK_GNSSSDR library provides more than 25 different kernels, for "8ic", "16ic" and "32fc" data types, each one with several proto-kernels, *i.e.*, specific implementations targeting a given SIMD technology, such as SSE2, SSE3, SSE4.1, AVX and AVX2 (aligned and unaligned memory versions), as well as NEON. Some of the most relevant for GNSS baseband processing were shown in Figure 1. The library was then integrated into GNSS-SDR, and the effectiveness of the proposed implementation was tested in four different computing platforms, with disparate processors, operating systems and compilers. Namely:

- **Platform #1 - Server**: a Dell's PowerEdge R730 server housing a CPU with two Intel Xeon E5-2630 v3 at 2.4 GHz (8 cores, 16 threads each) and an NVIDIA Tesla K10 GPU with 2 x 1536 CUDA cores clocked at 745 MHz. The operating system during tests was GNU/Linux Ubuntu 14.04, 64 bits, using GCC 4.9.2.
- **Platform #2 - Laptop**: Apple's MacBook Pro Late 2013, with an Intel Mobile Core i7-4558U (quad-core) CPU at 2.4 GHz (active cores can be speeded up to 3.8 GHz), and Hyper Threading technology allows the system to recognize eight total "cores" or "threads" (four real and four virtual), plus an NVIDIA GeForce GT 750M GPU with 384 CUDA cores clocked at 967 MHz. The operating system during tests was Mac OS X 10.11, using Apple LLVM / Clang version 7.0.2.
- **Platform #3 - Embedded development kit**: NVIDIA's Jetson TK1 developer kit, equipped with a quad-core ARM Cortex-A15 CPU at 2.32 GHz and an NVIDIA Kepler GPU with 192 CUDA cores clocked at 950 MHz. The operating system during tests was GNU/Linux Ubuntu 14.04, 32 bits, using GCC 4.8.4.
- **Platform #4 - Mini-computer**: Raspberry Pi 3 Model B, equipped with a Broadcom BCM2837 CPU (64 bit, ARMv8 quad-core ARM Cortex A53) clocked at 1.2 GHz. The operating system used during tests was Raspbian GNU/Linux 8 (jessie), 32 bits, using GCC 4.9.2.

This Section reports the results obtained by some of the key kernels implemented in the VOLK_GNSSSDR library and the performance achieved by the full software receiver in the described computer environments. All these results were obtained using GNSS-SDR v0.0.7 [23]. The reader is free to reproduce the experiments in his/her own machine by building that specific source code snapshot (or any other more recent version) and executing the provided profiling application `volk_gnsssdr_profile`, and is very welcome to contribute with bug reports, improvements and the addition of new kernels and proto-kernels.

7.1. SIMD acceleration

7.1.1. Results in x86 / x86_64 architectures

The acceleration factor achieved by different SIMD implementations with respect to the generic version, when executed in x86 / x86_64 processor architectures, are shown in Figures 4 and 5. Notably, the carrier removal and multiple correlation kernels for "16ic" and "32fc" data types (denoted as 16ic_rotator_dot_prod_16ic_xn and 32fc_rotator_dot_prod_32fc_xn) showed accelerations close to $\times 15$ and $\times 25$, respectively. Three correlators were used in those experiments, although the kernel interface admits an arbitrary number of them.

7.1.2. Results in ARM architectures

The acceleration factor achieved by NEON implementations with respect to the generic version, when executed in ARM-based computing platforms, are shown in Figures 6 and 7. Although, as stated in Section 5.3, NEON is not IEEE-754 compliant in its floating-point operations, numerical results show that they remain reasonably close to those obtained by IEEE-754 compliant technologies (*e.g.*, SSE) when input vectors are as long as $50,000$ complex items, which corresponds to the number of samples in 1 ms when sampled at 50 Msps in baseband, thus covering the GNSS signal with maximum available bandwidth (*i.e.*, Galileo E5).

7.2. Integration in a GNSS software receiver

In order to measure the performance of the parallelization strategies in combination with the data parallelization techniques described in this paper when applied to a full software-receiver, both the VOLK_GNSSSDR library and the GPU-targeted implementations were integrated into GNSS-SDR. The results obtained on the aforementioned computing platforms are shown below.

7.2.1. Number of channels processed in real-time

Figure 8 shows the execution time for different correlation vector lengths (2048, 4096, and 8192) and for different number of parallel channels targeting GPS L1 C/A signals and using three correlators per channel. This configuration mimicks typical receiver's configurations, and corresponds to the correlation lengths to be computed in 1 ms when sampling at 2.048, 4.096, and 8.192 Msps, respectively. In all platforms, `volk_profile` and `volk_gnsssdr_profile` were executed before the tests in order to enjoy the fastest available

SIMD implementation for each specific processor. Remarkably, ARM-based platforms achieved real-time processing of four or more channels in the narrowest bandwidth configuration.

7.2.2. GPU offloading

Fig. 9 shows the execution time for different correlation vector lengths (2048, 4096, and 8192) and for different number of parallel channels targeting GPS L1 C/A signals, with three correlators per channel. Results show that, as expected, the amount of available GPU resources limits the parallelization of correlation operations, hence defining the slope of the computing time growth. Platform #1 is equipped with the most powerful GPU, and thus it showed the best performance for all the tested correlation lengths. Regarding the real-time limit, defined at 1 ms, Table. 2 shows the maximum parallel channels available for each platform.

Table 2. Maximum number of real-time parallel channels for each platform using GPU accelerators.

Correlator length	2048	4096	8192
Platform #1	65	45	25
Platform #2	12	11	10
Platform #3	1	0	0

8. CONCLUSIONS

This paper described several parallelization techniques addressing computational efficiency, at different abstraction layers, and their specific application in the context of software-defined GNSS receivers. All those concepts were applied into a practical implementation available online under a free and open source software license. Building upon well-established open source frameworks and libraries, this paper showed that it is possible to achieve real-time operation in different computing environments. Portability was demonstrated by building and executing the same source code in a wide range of computing platforms, from high-end servers to tiny and affordable computers, using different operating systems and compilers, and showing notable acceleration factors of key operations in all of them.

As a practical outcome of the presented work, this paper introduced, to the best of authors' knowledge, the first free and open source software-defined GNSS receiver able to sustain real-time processing and to provide position fixes (as well as other GNSS data in form of RINEX files or RTCM messages streamed over a network) in ARM-based devices.

Future work will be related to the application of OpenMP for task parallelization; AVX+ technology using 512-bit registers and 64-bit NEON for data parallelization; and FPGA offloading in order to target real-time operation with higher signal bandwidths in multi-band, multi-constellation configurations.

ACKNOWLEDGEMENTS

This work has been developed in the frame of AUDITOR. This project has received funding from the European GNSS Agency under the European Unions Horizon 2020 research and innovation programme under grant agreement no. 687367. Authors ere also supported by the Spanish Ministry of Economy and Competitiveness through project TEC2015-69868-C2-2-R (MINECO/FEDER) and by the Government of Catalonia under Grant 2014–SGR–1567.

REFERENCES

[1] G. W. Heckler and J. L. Garrison, "SIMD correlator library for GNSS software receivers," *GPS Solutions*, vol. 10, no. 4, pp. 269–276, Nov. 2006, DOI: 10.1007/s10291-006-0037-5.

[2] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, The MIT Press, Cambridge, MA, 2008.

[3] N. P. Jouppi and D. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, Boston, MA, 1998, pp. 272–282.

[4] I. Bartunkova and B. Eissfeller, "Massive parallel algorithms for software GNSS signal simulation using GPU," in *Proc. of the 25th International Technical Meeting of The Satellite Division of the Institute of Navigation*, Nashville, TN, Sept. 2012, pp. 118–126.

[5] *GNSS-SDR. An Open Source Global Navigation Satellite Systems Software Defined Receiver*, Website: `http://gnss-sdr.org`. Accessed: January 30, 2017.

[6] *GNU Radio. The Free & Open Software Radio Ecosystem*, Website: `http://gnuradio.org`. Accessed: January 30, 2017.

[7] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, J. L. Rosenfeld, Ed., Stockholm, Sweden, Aug 1974, pp. 471–475, North Holland.

[8] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," in *Information processing*, B. Gilchrist, Ed., Amsterdam, NE, 1977, pp. 993–998, North Holland.

[9] T. M. Parks, *Bounded Scheduling of Process Networks*, Ph.D. thesis, University of California, Berkeley, CA, Dec. 1995.

[10] B. Huang, Z. Yao, F. Guo, S. Deng, X. Cui, and M. Lu, "STARx – A GPU based multi-system full-band real-time GNSS software receiver," in *Proc. of the 26th International Technical Meeting of The Satellite Division of the Institute of Navigation*, Nashville, TN, Sept. 2013, pp. 1549–1559.

[11] K. Karimi, A. G. Pamir, and H. Afzal, "Accelerating a cloud-based software GNSS receiver," *Intl. Journal of Grid and High Performance Computing*, vol. 6, no. 3, pp. 17–33, Jul./Sep. 2014, DOI: 10.4018/ijghpc.2014070102.

[12] L. Xu, N. I. Ziedan, X. Niu, and W. Guo, "Correlation acceleration in GNSS software receivers using a CUDA-enabled GPU," *GPS Solutions*, pp. 1–12, Available online since Feb. 2016, DOI: 10.1007/s10291-016-0516-2.

[13] Xilinx, San Jose, CA, *Zynq-7000 All Programmable SoC Overview DS190 (v1.9). Product Specification*, Jan. 2016.

[14] E. D. Kaplan, *Understanding GPS. Principles and Applications*, Artech House Publishers, 1996.

[15] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, "Automatic SIMD vectorization of Fast Fourier Transforms for the Larrabee and AVX instruction sets," in *Proc. 25th International Conference on Supercomputing*, Tucson, Arizona, May 31 - June 4, 2011, pp. 265–274.

[16] T. W. Rondeau, *Explaining the GNU Radio Scheduler*, Sep. 2013, Slides published online at `http://www.trondeau.com/blog`. Accessed: January 30, 2017.

[17] J. Corgan, "GNU Radio runtime operation," in *Proc. GNU Radio Conference*, Washington, DC, Aug. 24–28 2015, pp. 1–12.

[18] *Vector-Optimized Library of Kernels*, Website: `http://libvolk.org`. Accessed: January 30, 2017.

[19] T. W. Rondeau, N. McCarthy, and T. O'Shea, "SIMD programming in GNU Radio: Maintainable and user-friendly algorithm optimization with VOLK," in *Proc. of the Wireless Innovation Forum Conference of Wireless Communication Technologies and Software Defined Radio*, Washington, DC, Jan. 2013.

[20] N. West and D. Geiger, "Accelerating software radio on ARM: Adding NEON support to VOLK," in *Proc. IEEE Radio and Wireless Symposium*, Newport Beach, CA, Jan. 2014.

[21] *CUDA Programming Guide*, Website: `http://docs.nvidia.com/cuda/cuda-c-programming-guide`. Accessed: January 30, 2017.

[22] *NVIDIA CUDA Technology*, Website: `http://www.nvidia.com/CUDA`. Accessed: January 30, 2017.

[23] C. Fernández–Prades, J. Arribas, and L. Esteve, *GNSS-SDR v0.0.7*, Zenodo, May 2016, DOI : 10.5281/zenodo.51521. Available online at https://github.com/gnss-sdr/gnss-sdr/releases/tag/v0.0.7.

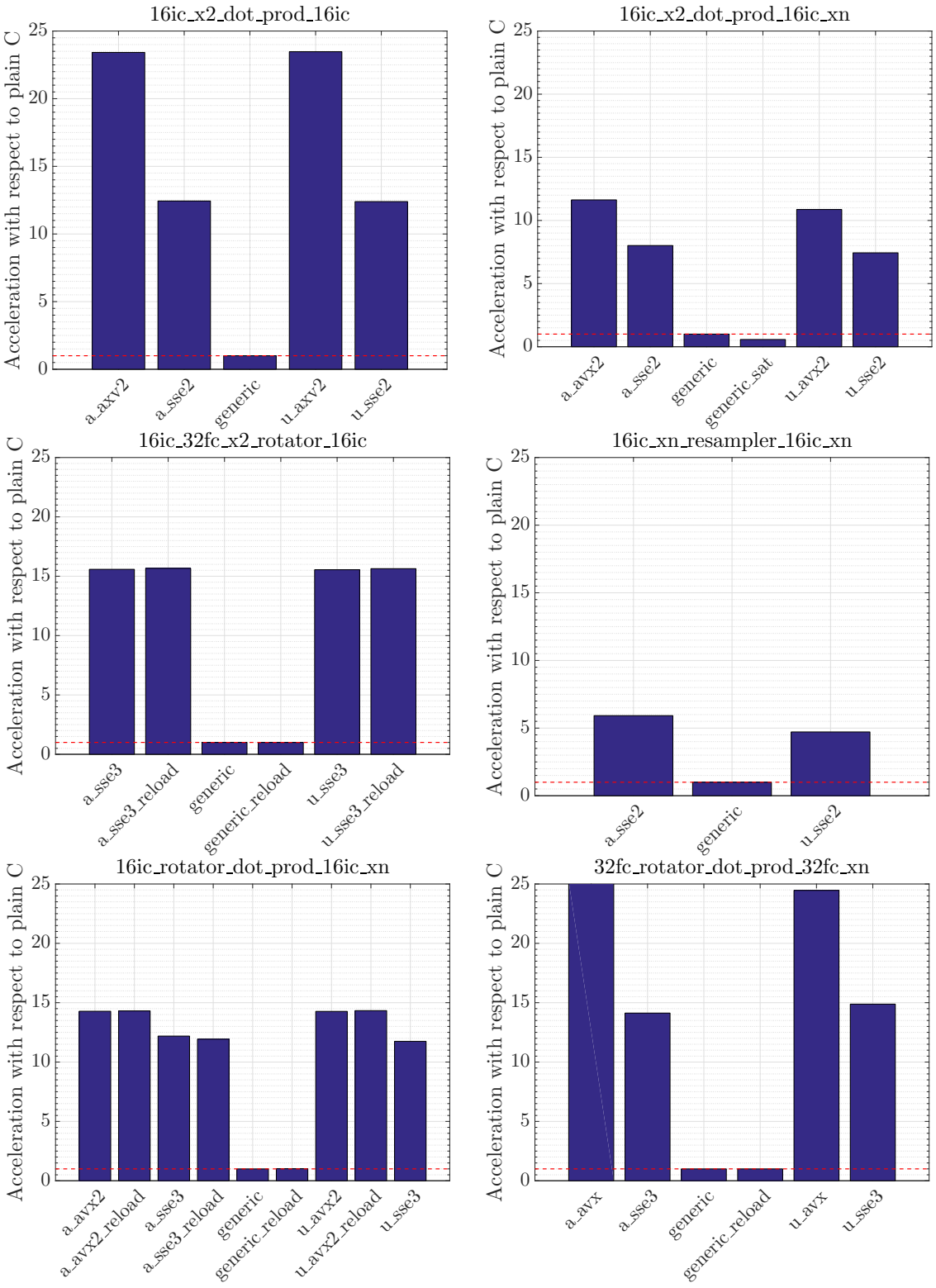

Fig. 4. Acceleration factor with respect to the generic implementation achieved by different proto-kernels in Platform #1. Operations were applied to vectors of 8111–item length, and the results were averaged over 1987 iterations.

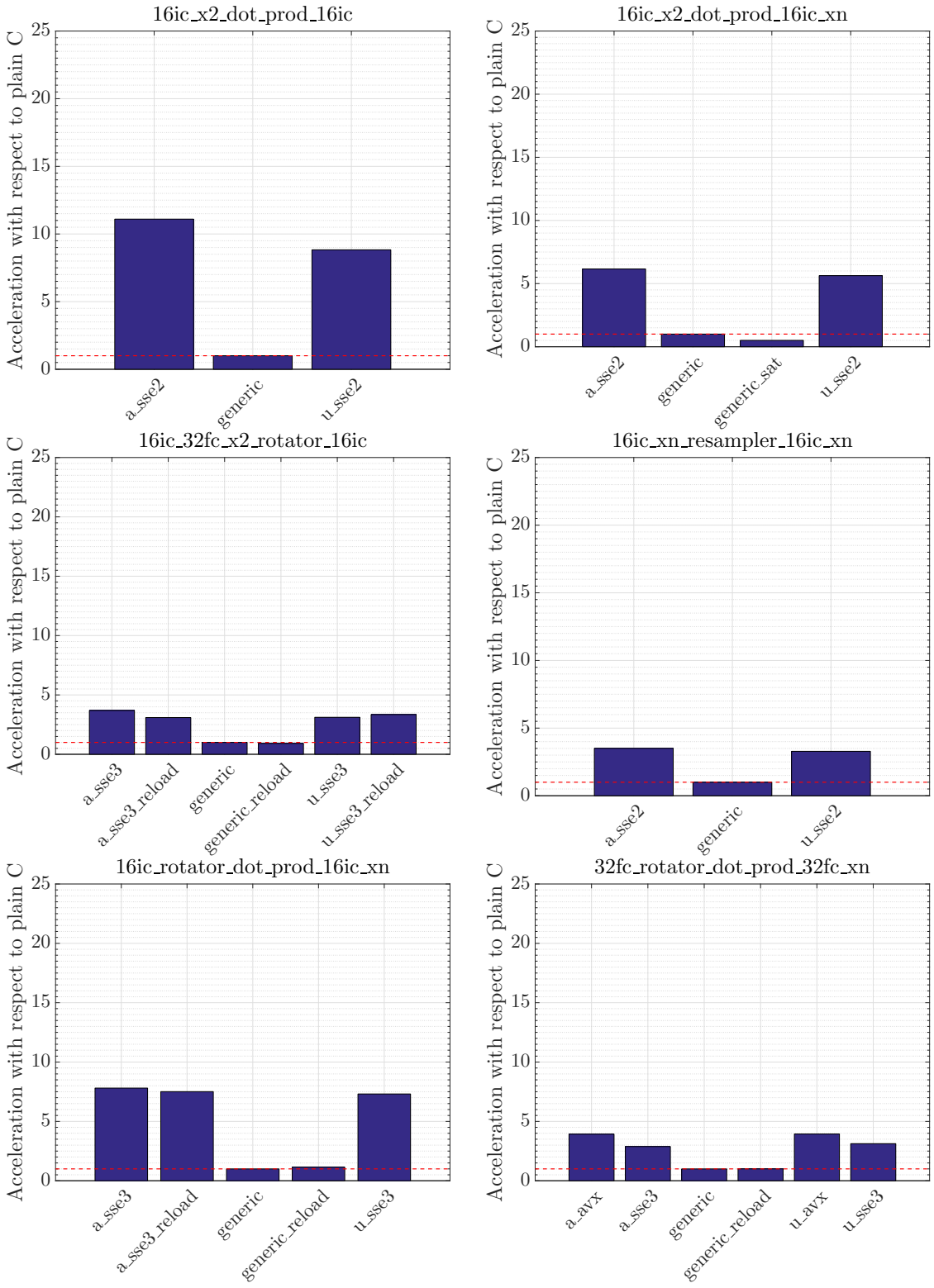

Fig. 5. Acceleration factor with respect to the generic implementation achieved by different proto-kernels in Platform #2. Operations were applied to vectors of 8111–item length, and the results were averaged over 1987 iterations.

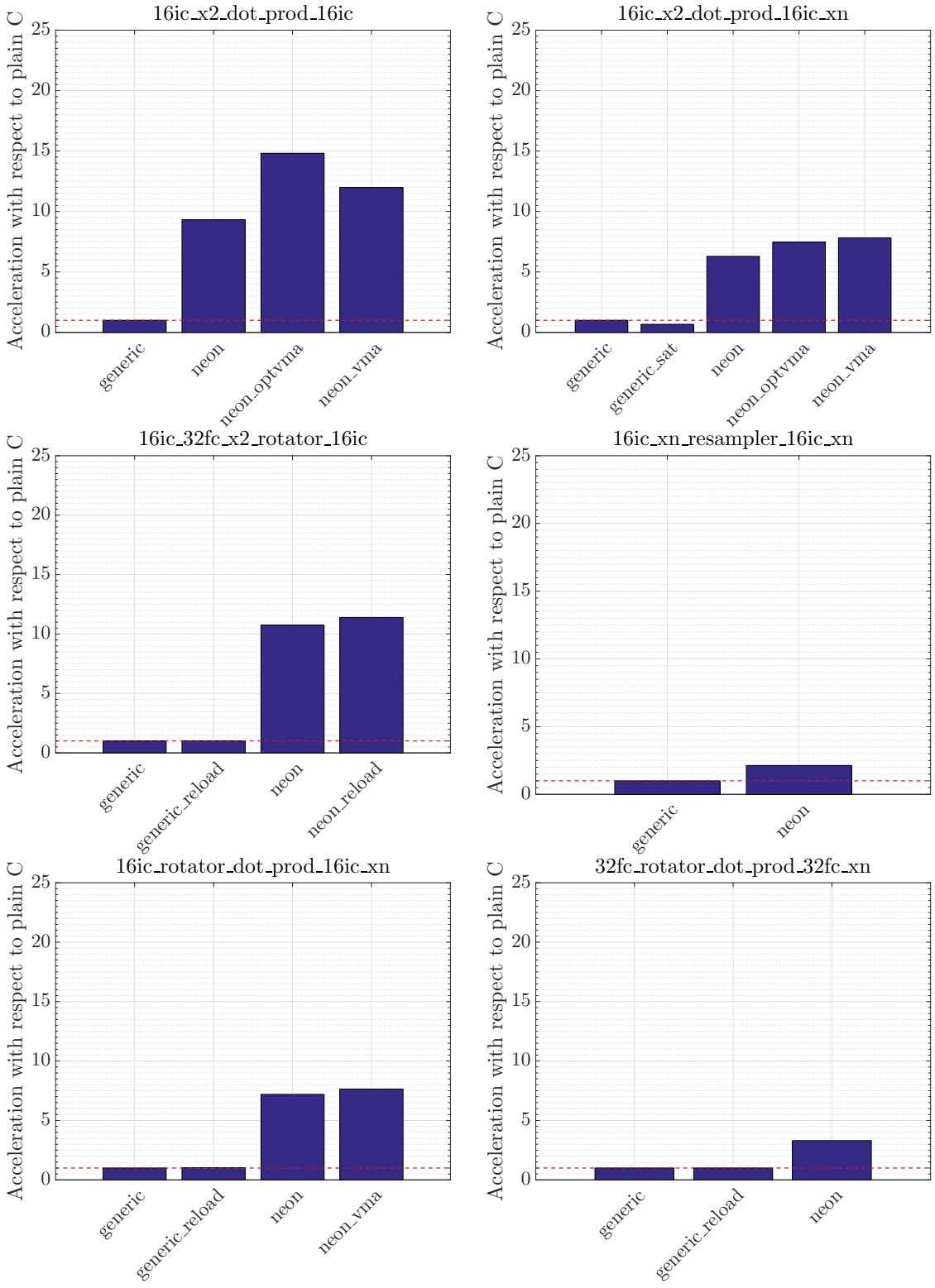

Fig. 6. Acceleration factor with respect to the generic implementation achieved by different proto-kernels in Platform #3. Operations were applied to vectors of 8111–item length, and the results were averaged over 1987 iterations.

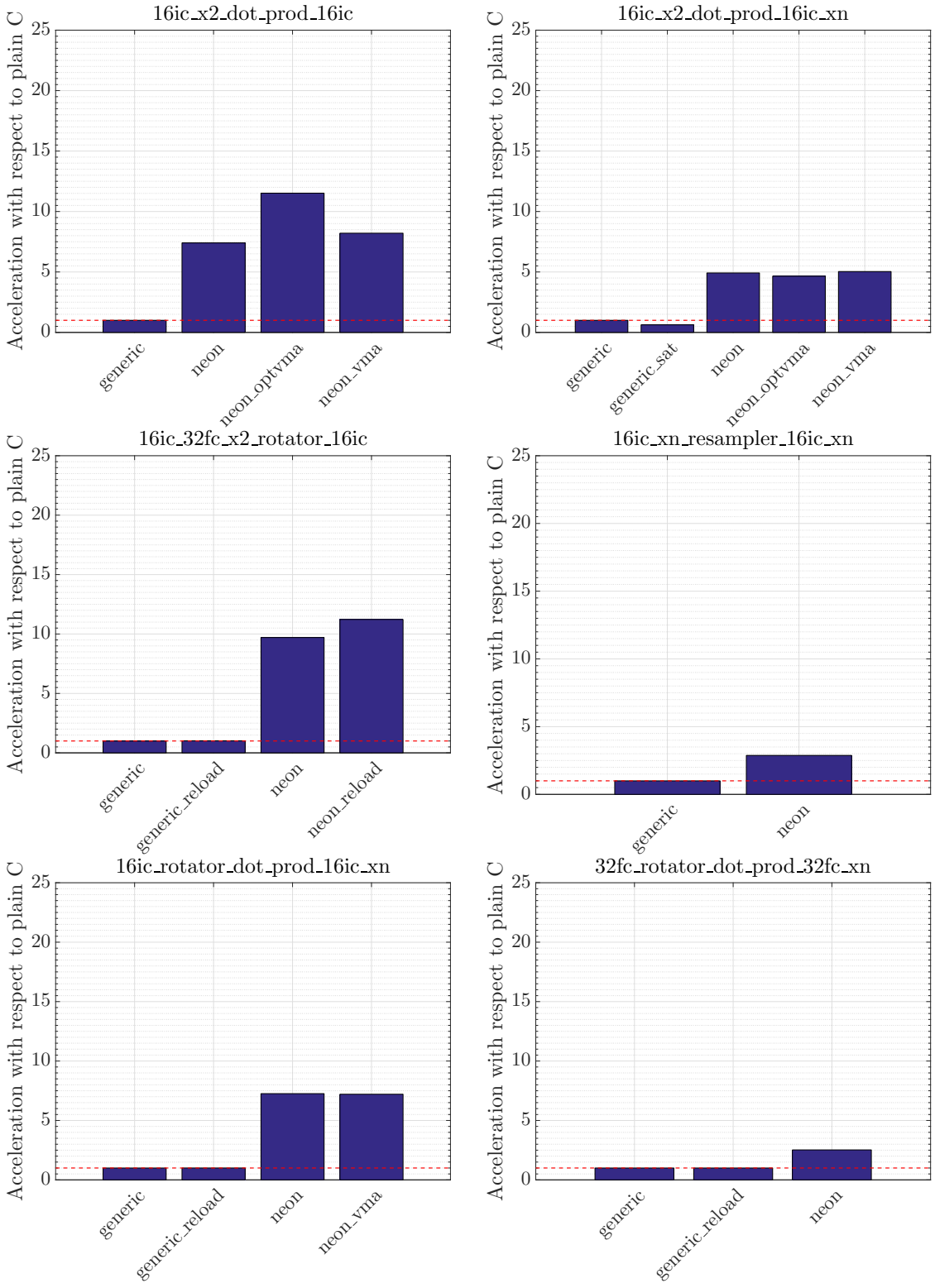

Fig. 7. Acceleration factor with respect to the generic implementation achieved by different proto-kernels in Platform #4. Operations were applied to vectors of 8111–item length, and the results were averaged over 1987 iterations.

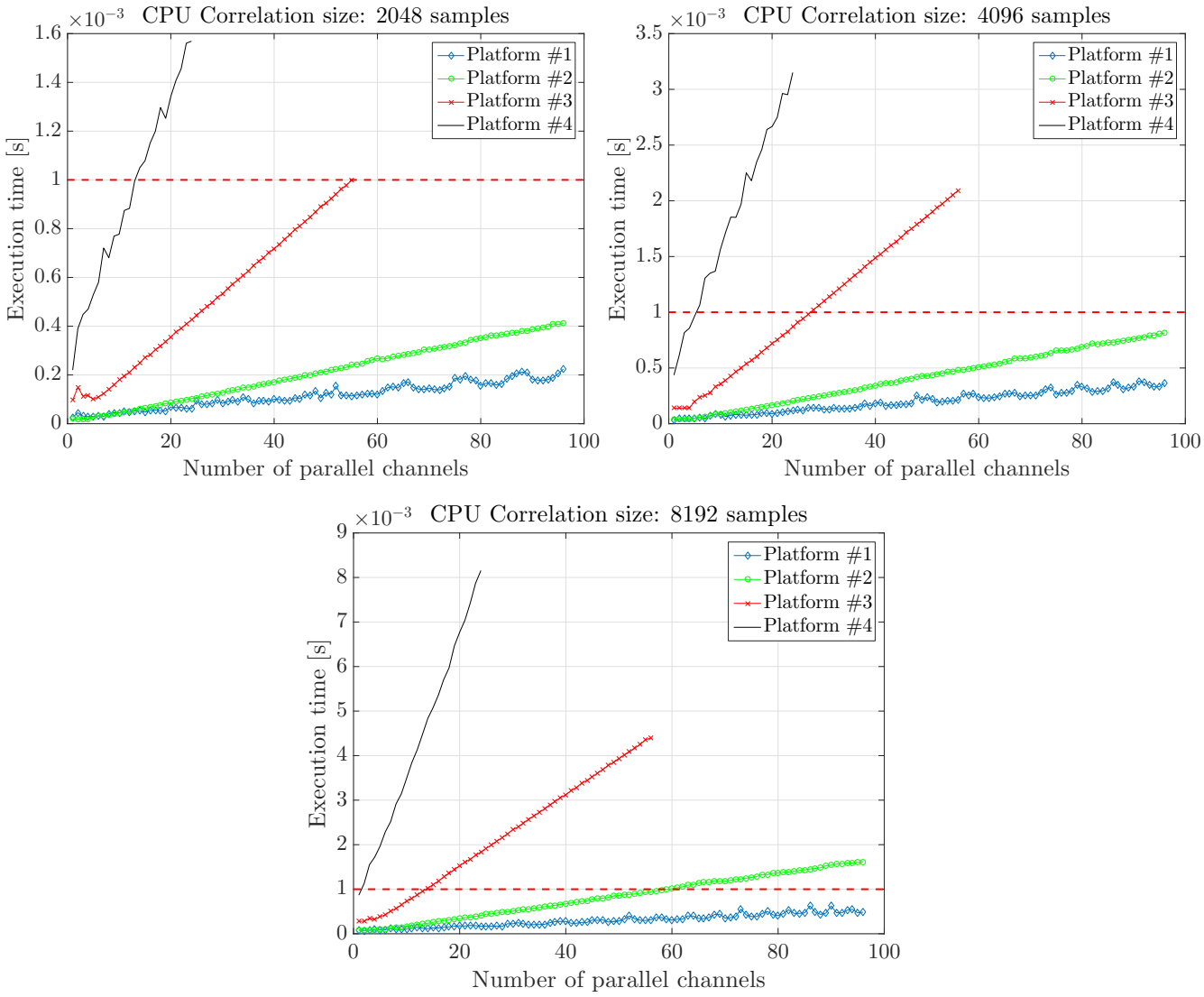

Fig. 8. CPU execution times (averaged for 1000 independent realizations) for different number of parallel channels and correlation lengths (2048, 4096 and 8192 samples of type "32fc") and executing platforms. Each channel was configured with 3 correlators. The intersection of these plots with the dashed red line at 1 ms indicates the number of channels that a given platform can sustain in real-time for GPS L1 C/A signals.

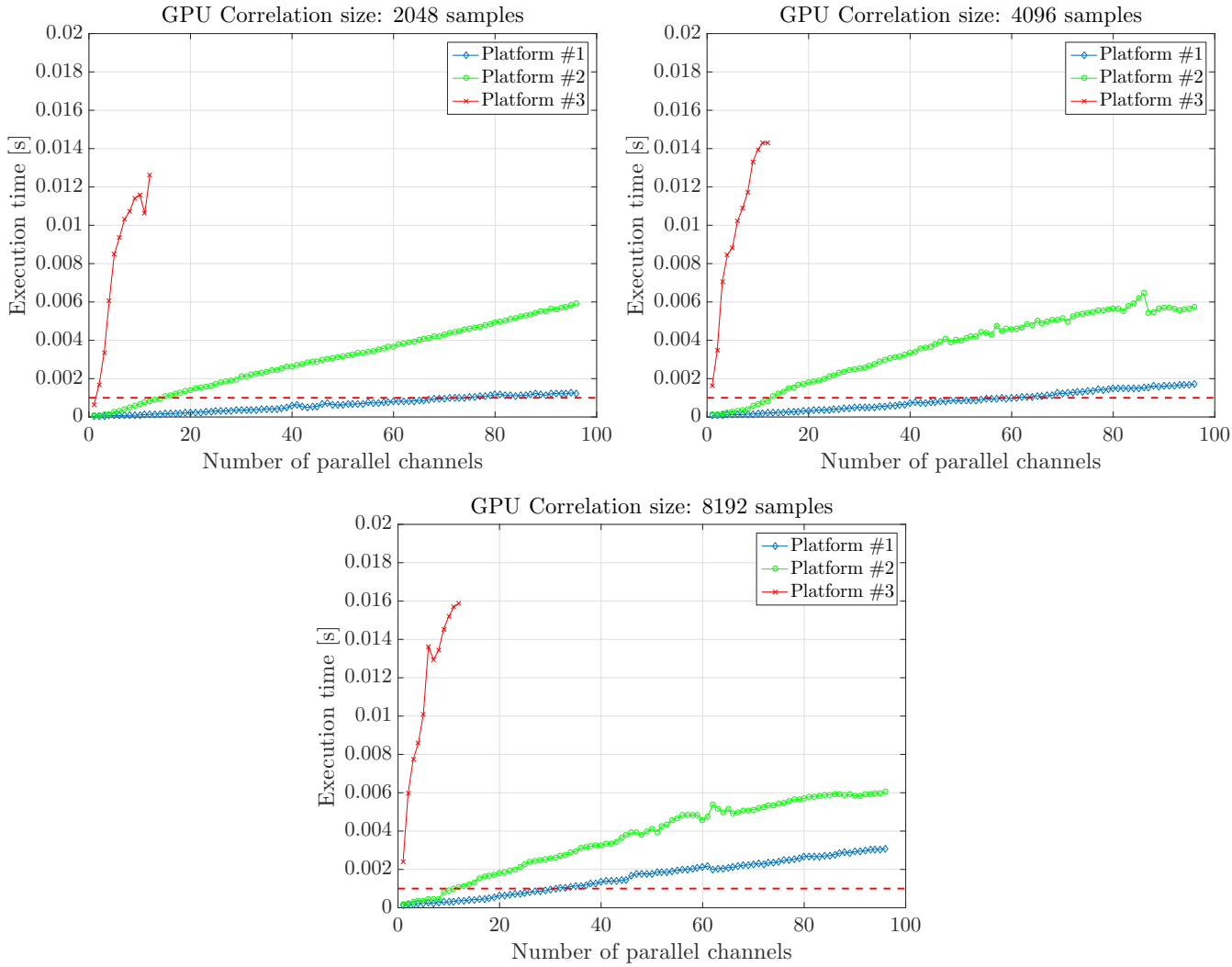

Fig. 9. GPU execution times (averaged for 1000 independent realizations) for different number of parallel channels and correlation lengths (2048, 4096 and 8192 samples of type "32fc") and executing platforms. Each channel was configured with 3 correlators. The intersection of these plots with the dashed red line at 1 ms indicates the number of channels that a given platform can sustain in real-time for GPS L1 C/A signals.