RESEARCH

FPGA-embedded Linearized Bregman Iteration algorithm for trend break detection

Felipe Calliari[1*] ⓘ, Gustavo Castro do Amaral[1,2] and Michael Lunglmayr[3]

*Correspondence:
felipe.calliari@opto.cetuc.puc-rio.br
[1]Center for Telecommunications Studies, Pontifical Catholic University of Rio de Janeiro, Marquês de São Vicente, Rio de Janeiro, 22451-900, Brazil
Full list of author information is available at the end of the article

Abstract

Detection of level shifts in a noisy signal, or trend break detection, is a problem that appears in several research fields, from biophysics to optics and economics. Although many algorithms have been developed to deal with such a problem, accurate and low-complexity trend break detection is still an active topic of research. The Linearized Bregman Iterations have been recently presented as a low-complexity and computationally efficient algorithm to tackle this problem, with a formidable structure that could benefit immensely from hardware implementation. In this work, a hardware architecture of the Linearized Bregman Iteration algorithm is presented and tested on a Field Programmable Gate Array (FPGA). The hardware is synthesized in different-sized FPGAs, and the percentage of used hardware, as well as the maximum frequency enabled by the design, indicate that an approximately 100 gain factor in processing time, concerning the software implementation, can be achieved. This represents a tremendous advantage in using a dedicated unit for trend break detection applications. The proposed architecture is compared with a state-of-the-art hardware structure for sparse estimation, and the results indicate that its performance concerning trend break detection is much more pronounced while, at the same time, being the indicated solution for long datasets.

Keywords: Linearized Bregman Iterations, Trend break detection, FPGA

1 Introduction

Trend break detection in the presence of noise is a broad problem that can be found across different research fields [1–4]. For that reason, several different methodologies have been proposed in the literature [5–7], with the ones that make use of ℓ_1 regularization to counter the problem's inherent high-dimensionality arguably figuring as the most successful ones [8, 9]. Such an approach is required for highly reliable estimation results [7]. Even though such regularization allows the problem to be solved in a computationally efficient manner (usually associated to a complexity which is proportional to a polynomial function of the number of inputs), the fact that a computer can solve the problem does not necessarily mean that the result is achieved quickly, practically speaking. In certain contexts, achieving elapsed algorithm times in the order of seconds as opposed to minutes may yield a substantial impact on the application [10].
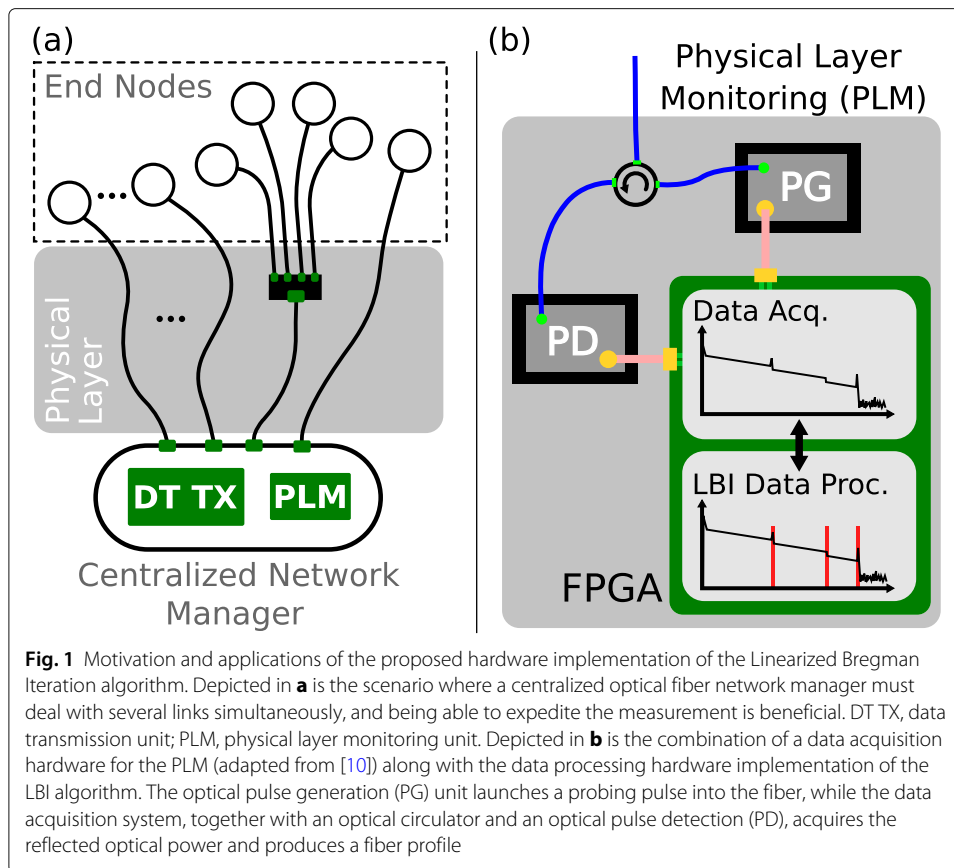
It is a widespread notion that certain problems, despite their complexity, may be accelerated depending on the implementation; parallel programming, in which several parts of the same procedure are processed independently and simultaneously, is one of the most celebrated examples [11]. Field Programmable Gate Arrays (FPGAs) are extremely versatile hardware structures that offer the following [12–14]: great flexibility to design high-speed high-density digital hardware, easiness of programmability and reconfiguration, energy efficiency, high resource utilization, low cost, and the possibility to combine parallel processing structures with serial control units. FPGAs have been used as a versatile computing platform accelerating algorithms through dedicated and carefully designed architectures in a wide range of fields [15] such as cryptography [16], image processing [17], and machine learning [18].

1.1 Motivation and contributions

Recently, Linearized Bregman Iterations (LBI), a class of implementation-efficient and low-complexity algorithms, have been presented as an extremely attractive solution for trend break detection [1]. There, it was shown that the LBI algorithm outperforms the classical LASSO solution in the specific problem of trend break detection for fiber fault analysis. In this case, the better performance can be attributed to two factors. First, it is well known that combined L1/L2 regularization terms can handle problems where estimation vectors with correlated elements are expected better than the LASSO solution [19] and the problem at hand is expected to have such correlations. Second, the strict convexity when using a combined L1/L2 term in the cost function has been reported to improve the convergence behavior of sparse estimation algorithms [20]. Furthermore, both the structures of the trend break detection problem and of the LBI algorithm allow for simple hardware units, relying mainly on adders and efficient memory management, to conduct the core procedure, thereby avoiding hardware-complex multiplication and division operations [21]. In [1], the focus is on the detection of trend breaks associated to fiber faults: to ensure robustness of network operation, detection of such faults must be remedied as fast as possible; this, in turn, creates a demand for highly reliable and fast trend break detection results, since a network operator might have several thousand optical fibers under his responsibility [22, 23], and thus, the faster data associated with one link can be processed, the faster information about all links will be available.

The possibility of increasing the time efficiency of the algorithm due to its hardware implementation is of great interest in this context, as pictorially presented in Fig. 1a. Furthermore, as the measurement is often done by stand-alone Optical Time-Domain Reflectometry (OTDR) devices, the eventual goal is to implement the processing directly in such a device, where FPGAs are usually employed. Achieving an FPGA implementation of the detection algorithm is, then, preferable, since it would allow data acquisition and processing to be performed in the same embedded device, as pictorially presented in Fig. 1b.

In this work, the hardware implementation of the LBI algorithm is studied in depth and is simulated and synthesized for different FPGAs. A novel hardware architecture is presented, and its main processing units are discussed. VHDL simulation environments enable a step-by-step comparison and validation of the processing stages referenced by the computer algorithm implementation [1]. Hardware synthesis results allow determination of both device usage with different FPGA sizes and maximum clock frequency; the

Fig. 1 Motivation and applications of the proposed hardware implementation of the Linearized Bregman Iteration algorithm. Depicted in **a** is the scenario where a centralized optical fiber network manager must deal with several links simultaneously, and being able to expedite the measurement is beneficial. DT TX, data transmission unit; PLM, physical layer monitoring unit. Depicted in **b** is the combination of a data acquisition hardware for the PLM (adapted from [10]) along with the data processing hardware implementation of the LBI algorithm. The optical pulse generation (PG) unit launches a probing pulse into the fiber, while the data acquisition system, together with an optical circulator and an optical pulse detection (PD), acquires the reflected optical power and produces a fiber profile

latter, combined with the average number of clock cycles per iteration loop, makes total processing time calculation possible for different problem instance sizes. A reduction factor on the elapsed algorithm time of approximately 100 is achieved, which represents a substantial upgrade and warrants usage of dedicated hardware for trend break detection. The main overall contributions of this work are discriminated below:

- Speed-ups of the order of 100 times in the hardware implementation with respect to the software implementation of the LBI algorithm for trend break detection. The speed-ups are solely attributed to the hardware architecture and the usage of parallel-pipelined arithmetic units (such as the so-called pipelined adder tree) and to efficient memory organization and control, since no loss of performance is identified between hardware and software implementations.
- A comprehensive analysis of the hardware architecture, including the main processing and control units that allow for the algorithm structure to be implemented in hardware. This analysis provides a straightforward means to determining the total number of clock cycles necessary for the algorithm to elapse.
- A software implementation of the algorithm that has been validated bit-wise with respect to the hardware implementation, thereby allowing for the results of the proposed implementation to be verified in a simulated environment.
- Comparison, with respect to both performance and hardware characteristics, of the proposed hardware implementation with state-of-the-art hardware implementations

of the Orthogonal Matching Pursuit algorithm, which is used in the context of sparse estimation and finds widespread interest in the current literature.

The paper is divided as follows. In Section 3, a brief review of the LBI algorithm for trend break detection is performed, including the structure of the candidate matrix and the pseudocode based on which the hardware architecture is developed. Section 4 presents the digital hardware architectural concept as well as focused descriptions of its main units; the estimated number of clock cycles until the algorithm elapses is derived based on this architecture. In Section 5, comparative results between the simulated hardware implementation and its Julia-based software counterpart are discussed. Synthesis parameters for two target FPGAs (ALTERA CYCLONE V and ALTERA STRATIX V) are also reported. Case studies (both for a real-world fiber profile and simulated data series) are discussed in Section 6.1. Section 6.2 is dedicated to the comparative analysis between the proposed architecture and a state-of-the-art algorithm in the context of trend break detection. Finally, Section 7 concludes the paper.

2 Methods

The aim of this work was to develop and evaluate a hardware architecture to accelerate sparse estimation for reliably solving the trend break detection problem. To evaluate the capabilities of the proposed architecture, the following methods have been used:

- A commercially available state-of-the-art synthesis software (INTEL QUARTUS PRIME) was used to evaluate the maximum clock frequency and the device occupation of the design for different sizes (i.e., the number of used block RAMs in parallel).
- An analytical description was developed and used (validated by hardware simulation) to calculate the number of clock cycles necessary to perform the estimation tasks.
- A bit-true simulation environment was developed, and after bit-true validation against a commercially available hardware simulation tool (Mentor Graphics Modelsim), it was used to evaluate the performance of the proposed architecture in large-scale simulation studies.
- The bit-true simulation results of the hardware implementation are compared to double precision floating point results in terms of their averaged squared error norms.
- A real-world dataset measured in the lab by an Optical Time-Domain Reflectometry measurement device has been used to evaluate the algorithm, both in double precision floating point as well as in its bit-true quantized version used in hardware.
- The LBI algorithm discussed in this work is compared to the OMP algorithm based on simulation studies and the thereby calculated figures of merit: the precision and the Matthews correlation coefficient.
- The LBI algorithm and the OMP algorithm are compared based on their computational complexity analysis.

3 The Linearized Bregman Iteration algorithm for trend break detection

Under the assumption that the trend break detection problem is a sparse one, i.e., the number of candidate vectors that describe the signal of interest is much smaller than the number of observations, it can be cast into the combined ℓ_1/ℓ_2 problem of the form [1]:

$$\min_{\boldsymbol{\beta}} \lambda ||\boldsymbol{\beta}||_1 + \frac{1}{2}||\boldsymbol{\beta}||_2^2 \ \ s.t. \ \ \mathbf{A}\boldsymbol{\beta} = \mathbf{y}, \tag{1}$$

where $\mathbf{A}$ is the dictionary, with each candidate vector stored in a column; $\boldsymbol{\beta}$ is the vector containing the coefficients of the weighted linear combination of dictionary vectors that will approximate the signal of interest represented by the data series $\mathbf{y}$; and λ is a parameter that adjusts the weight of the ℓ_1 versus the ℓ_2 norms in the cost function. Adaptation of the Linearized Bregman Iteration algorithm to trend break detection has been presented in [1] in a context where a linear trend is also expected in the signal of interest. In order to simplify and generalize the implementation, this linear trend is not considered in the current implementation. Incorporating the linear trend in the proposed architecture is, however, straightforward.

Throughout the manuscript, the length, in data points, of the signal of interest $\mathbf{y}$ will be defined as N, i.e., $\mathbf{y}$ and $\boldsymbol{\beta}$ are N-dimensional vectors and $\mathbf{A}$ is an $N \times N$ matrix. The Linearized Bregman Iteration algorithm has a cyclic structure, involving, in a single iteration, an approximate gradient descent (AGD) followed by a non-linear shrink function of the form: $\mathrm{shrink}\left(\mathbf{v}_j, \lambda\right) = \max\left(|\mathbf{v}_j| - \lambda, 0\right) \cdot \mathrm{sign}\left(\mathbf{v}_j\right)$ [24]. Due to the special structure of the candidate dictionary matrix $\mathbf{A}$ for the trend break detection problem, namely:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & 1 & \cdots & 1 & 0 \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}, \tag{2}$$

its storage is not necessary for the AGD calculation, as the latter can be rewritten as:

$$\begin{aligned} \mathbf{v}^{(i+1)} &= \mathbf{v}^{(i)} + \frac{\mathbf{a}_k}{||\mathbf{a}_k||_2^2}\left(\mathbf{y}_k - \mathbf{a}_k^{\mathrm{T}}\boldsymbol{\beta}^{(i)}\right) \\ &= \mathbf{v}^{(i)} + \frac{\mathbf{a}_k}{||\mathbf{a}_k||_2^2}\left(\mathbf{y}_k - \sum_{s=1}^{k+1}\beta_s\right) \end{aligned} \tag{3}$$

where the $\mathbf{a}_k$ represent rows of the candidate matrix, the superscripted i represents the iteration index, and the index $k \in [1 : N]$ controls the cyclic re-use of rows of $\mathbf{A}$ as the iteration index evolves, i.e., $k = \mathrm{mod}\left((i-1), N\right) + 1$.

The $\mathbf{a}_k$, in turn, have an interesting structure that allows the AGD to be further optimized and the calculation to be performed only for those indices where $\mathbf{a}_{k,j} \neq 0$. In other words (and also considering the fact that $||\mathbf{a}_i||_2^2 = k$),

$$\mathbf{v}_j^{(i+1)} = \begin{cases} \mathbf{v}_j^{(i)} + \frac{1}{k}\left(\mathbf{y}_k - \sum_{s=1}^{k+1}\boldsymbol{\beta}_s\right), & a_{k,j} = 1 \\ \mathbf{v}_j^{(i)}, & a_{k,j} = 0 \end{cases}, \tag{4}$$

which, considering computational implementation, translates into accessing and manipulating only those values of vector $\mathbf{v}^{(k)}$ up to index j. A final observation of the structure of matrix $\mathbf{A}$ (namely, the fact that it is a square matrix) reveals that a single index k is sufficient to control an iteration of the algorithm. The resulting procedure, presented as pseudocode in Algorithm 1, efficiently solves the trend break detection problem with low memory usage.

For the presented algorithm, on average, $(N-1)/2$ additions per iteration are required. This can be seen from Algorithm 1: the first iteration requires 1 addition, the second 2, and the last N; culminating in $N \cdot (N+1)/2$ additions for N iterations. It is interesting to note that a single multiplication operation is necessary after the result of the summation is performed, which greatly simplifies the procedure and avoids an overload of complex arithmetic structures. As it has been shown in [1], Linearized Bregman Iterations lead to less complex algorithms than alternative approaches such as the adaptive LASSO [7]. Using an FPGA, one could speed up the additions thereby speeding up a single iteration, resulting in a speed-up of the whole algorithm. Indeed, as is shown here, this solution allows the core algorithm to be processed within a fraction of the time it would take on a high-end server processor.

3.1 Hardware implementation considerations of the LBI algorithm

3.1.1 The ordinary least squares final step

It is important to note that Algorithm 1 is an adaptation of the pseudocode presented in [1], where only the computation-heavy part of the procedure is depicted. Its purpose is to identify the relevant non-zero values of the $\hat{\boldsymbol{\beta}}$ vector that compose the output or, in other words, reduce the dimension of the detection space focusing on the subspace spanned by the relevant candidate vectors. After this procedure, it is usual to perform an ordinary least square (OLS) in this reduced subspace in order to remove any biasing introduced by the algorithm; operating on the reduced subspace found by the LBI drastically reduces the complexity of the OLS. This step, which involves matrices transposing and inverting, can be efficiently conducted in a standard personal computer, and even though this could also be implemented in the same hardware structure that contains the core algorithm [1], the goal of this work is to present the latter and the OLS step is left as a post-processing step to be performed in a different processing unit.

With respect to the OLS post-processing, interesting functionalities of the so-called Systems On Chip (SOC), which combine FPGAs with embedded CPUs as, for example,

---

Algorithm 1 Linearized Bregman Iteration for Trend Break Detection

Require: Measurement vector $\mathbf{y}$, λ, $\boldsymbol{\beta}_{\text{start}}$, $\mathbf{v}_{\text{start}}$, L

Ensure: Estimated $\hat{\boldsymbol{\beta}}$

1: $\boldsymbol{\beta}^{(0)} \leftarrow \boldsymbol{\beta}_{\text{start}}$
2: $\mathbf{v}^{(0)} \leftarrow \mathbf{v}_{\text{start}}$
3: $i \leftarrow 1$
4: **while** $i < L$ **do**
5:      $k \leftarrow \text{mod}((i-1), N) + 1$ ▷ cyclic re-use of rows of $\mathbf{A}$
6:      $\mu_k \leftarrow \frac{1}{k}$
7:      $e \leftarrow \left(\mathbf{y}_k - \sum_{s=1}^{k+1} \beta_s^{(i)}\right)$
                                                                  ▷ instantaneous error with inner product
8:      $d \leftarrow \mu_k e$
9:      **for** $j = 1..k$ **do**
10:          $\mathbf{v}_j^{(i+1)} \leftarrow \mathbf{v}_j^{(i)} + d$
11:          $\boldsymbol{\beta}_j^{(i+1)} \leftarrow \text{shrink}\left(\mathbf{v}_j^{(i+1)}, \lambda\right)$
12:      **end for**
13:      $i \leftarrow i + 1$
14: **end while**

in the used CYCLONE V, can be harnessed for this goal. In fact, they enable the delegation of the reduced subspace OLS problem to the embedded CPU. Measurements have been performed on the SOC contained in the CYCLONE V running Linux (Ubuntu 16.04) on its dual-core arm processor. For typical OLS instance sizes of 15,000×20 (a reasonable upper-bound according to [1] for a 15,000-sample fiber profile), a run-time of about 0.15 s in the Julia language has been obtained. It is noteworthy that, in contrast, running the LBI core algorithm (typically requiring 100–200 s on a much more powerful XEON Server processor) is practically not feasible on such an embedded CPU. However, as the measurement results of the OLS problem show, and due to its negligible time overhead, delegating the OLS post-processing step to an embedded CPU is indeed feasible. Once again, even though the ordinary Kaczmarz algorithm approximately solves the OLS problem and can re-use blocks from the sparse Kaczmarz, on which the LBI algorithm is based, thereby allowing the implementation of both in the same FPGA chip, this adaptation would require several design add-ons, which are not the focus of this research.

In summary, the reason to presently delegate the OLS step to a CPU is threefold: first, a CPU can easily deal with the problem in the reduced subspace identified by the core procedure of the LBI; second, it allows this work to focus on the core structure of the LBI algorithm; and finally, since an embedded CPU might be already available (as is the case of the CYCLONE V studied here), performing the OLS on the CPU of a SOC might be the more practical approach.

3.1.2 Scaling and arithmetic dynamic range

Also left as a pre-processing step is the scaling of the data vector $\mathbf{y}$, which is necessary to ensure the correct behavior of step 7 in Algorithm 1 when using the 20-bit fixed-point format—this seemly arbitrary value will be clarified in Section 6.1 utilizing limited arithmetic dynamic range. In other words, one must make sure that no overflows of the arithmetic dynamic range are observed when performing the summation of $\boldsymbol{\beta}$ values. The scaling is intimately connected to the available arithmetical dynamic range, which, in turn, is connected to the memory resources of the FPGA board, thereby constituting a design-related compromise relationship. In case of overscaling, the arithmetic dynamic range will be hindered; to overcome this, a higher number of bits can be assigned to the data points, which then increases the resources necessary in the FPGA board. On the other hand, in case of underscaling, the results may overflow, creating errors that can jeopardize the algorithm's convergence. A scaling factor consistent with the algorithm's convergence can be determined according to the following considerations.

The major source for overflows is the sum calculation of $\boldsymbol{\beta}$ values in line 7 of Algorithm 1. Empirical tests conducted based on the testbench developed in [1] indicated that a scaling based on dividing the data vector $\mathbf{y}$ by its maximum value allowed to obtain the results shown in Section 5 without harming overflow effects. This is due to the firmly non-expansive property [25] of the shrink function as well as the negative feedback of the error between $\sum_{s=1}^{k+1} \boldsymbol{\beta}_s^{(i)}$ and y_k (line 7 of Algorithm 1). To clarify the negative feedback effect, one could multiply both sides of line 7 by -1, i.e., $-e = \mathbf{y}_k - \sum_{s=1}^{k+1} \boldsymbol{\beta}_s^{(i)}$. This procedure would require μ_k, in line 10, to also be multiplied by -1.

The algorithm can, then, be interpreted as a stabilizing loop on the values of $\mathbf{v}_j^{(i)}$ with the mentioned negative feedback on the deviation between the sum of $\boldsymbol{\beta}$ values (functions of $\mathbf{v}_j^{(i)}$) and the corresponding $\mathbf{y}_k$, which causes overshoots of the sum of $\boldsymbol{\beta}$ values

to be immediately corrected in the next iterations. This leads to the fact that, even in worst case scenarios (multiple up and down trend breaks in the measurements), the sum of β values scarcely goes above unit, considering the abovementioned normalization procedure. Moreover, even though its ratio of occurrence is negligible, in the case an overshoot occurs, the excess value would be small thereby not compromising the convergence of the algorithm, as the performance of the quantized version in Secttion 5 demonstrates. The closeness of these results to the ones obtained using double precision floating point shows that, practically, harming effects due to overflow can be neglected.

4 FPGA architecture

An inherently iterative algorithm and, thus, sequential in nature, the LBI cannot have its iterations parallelized. The operations within an iteration, on the other hand, could greatly benefit from parallelization. The challenge is, thus, to design an architecture that allows high parallelism but that still keeps the effort for control logic, as well as the datapath, manageable. Parallelism in digital hardware benefits not only from parallel calculation units but also, and sometimes more crucial, from an architecture that efficiently feeds the required data to the calculation units in a parallel fashion. Although today's FPGAs typically provide a considerable number of block RAMs (BRAMs), these are implemented in such a way that the number of BRAMs is reduced in exchange for extensive individual memory depth as it is beneficial for many applications.

For the current architecture, this represents a trade-off: while the values of β and $\mathbf{v}$ benefit from parallel access, the values of y are preferably accessed in sequential order, and at the same time, using dedicated BRAMs for $\mathbf{y}$ would limit the scalability and flexibility of the algorithm. For this reason, a combined parallel adder tree and parallel multiplexer tree architecture have been implemented for convenient parallel access to the estimation variables (β and $\mathbf{v}$) involved in the core calculations of the algorithm as well as efficient data routing of the values of $\mathbf{y}$.

As already mentioned, even though the iterative nature of the Linearized Bregman Iteration algorithm does not allow for parallelization over the iterations, two core operations that permit parallel pipelining can be identified within a single iteration, as presented in Algorithm 1: the summation of k entries of the vector β and the processing (including update, shrinkage, and storage) of vectors $\mathbf{v}$ and β. By instantiating parallel memory structures, both operations that represent computational bottlenecks of the algorithm's iterations can be optimized. On the one hand, the summation can be efficiently performed in a so-called *parallel adder tree* (PAT) (logarithmic number of time steps) given that the data can be accessed in parallel. On the other hand, parallel processing of the data in vectors $\mathbf{v}$ and β can also be accelerated if load/storage can be performed in parallel. Since the algorithm relies on the computation of several iterations to converge, optimizing these two procedures allows for substantial gains in processing time.

4.1 Memory structure

In order to harness the parallel speed-up of the PAT, the entries of vector β must also be accessed in parallel, which can be accomplished through the instantiation of parallel block RAMs (BRAMs). The data storage is structured as follows:

$$
t \Bigg\uparrow \begin{array}{ccc} \text{BRAM}(1) & \text{BRAM}(2) & \text{BRAM}(M) \\ \left[\begin{array}{cccc} \boldsymbol{\beta}[\,TM+1] & \boldsymbol{\beta}[\,TM+2] & \cdots & \boldsymbol{\beta}[\,TM+M] \\ \vdots & \vdots & \cdots & \vdots \\ \boldsymbol{\beta}[\,2M+1] & \boldsymbol{\beta}[\,2M+2] & \cdots & \boldsymbol{\beta}[\,2M+M] \\ \boldsymbol{\beta}[\,M+1] & \boldsymbol{\beta}[\,M+2] & \cdots & \boldsymbol{\beta}[\,M+M] \\ \boldsymbol{\beta}[\,1] & \boldsymbol{\beta}[\,2] & \cdots & \boldsymbol{\beta}[\,M] \end{array} \right] \end{array} ,
\tag{5}
$$

$$\xrightarrow{\quad m \quad}$$

where M is the number of parallel BRAMs available in the FPGA. In such a structure, a single arbitrary BRAM, say m, will contain the entries:

$$
[\,m + tM] \, \forall t \in [\,0; T\,], \, m \in [\,1; M\,] : T = \left\lceil \frac{N}{M} \right\rceil ,
$$

where the ceiling operator is denoted by $\lceil \cdot \rceil$.

Vector $\boldsymbol{\beta}$, however, is not the only vector stored throughout processing: vectors **y** and **v** are also necessary. Since all these contain the same number N of entries, the data is sectioned such that the address depth of each BRAM is divided into three slices with address pointers (ap) associated with $\boldsymbol{\beta}$ ($\boldsymbol{\beta}_{\text{ap}}$), **y** ($\mathbf{y}_{\text{ap}}$), and **v** ($\mathbf{v}_{\text{ap}}$); $\boldsymbol{\beta}_{\text{ap}}$ is arbitrarily set to zero. Under this rationale, entries of vectors **v** and **y** would appear at addresses $t + \mathbf{v}_{\text{ap}}$ and $t + \mathbf{y}_{\text{ap}}$, respectively, even though, for simplicity, only entries of vector $\boldsymbol{\beta}$ are shown in Eq. 5. Using this data storage structure, all positions $[\,tM + 1 : tM + M]$ of either vectors can be accessed from parallel BRAMs within a clock cycle; such data segment will henceforth be referred to as a *parallel row*, with t the *parallel row* pointer following its definition in Eq. 5. A block diagram of the digital hardware architecture depicting a single BRAM and including the major structures of the LBI algorithm hardware implementation is presented in detail in Fig. 2.

Apart from the PAT, a Pipelined Multiplexer Tree (PMT) is used to select the specific value of the data series **y**, namely $\mathbf{y}(k)$, from which the result of the partial summation
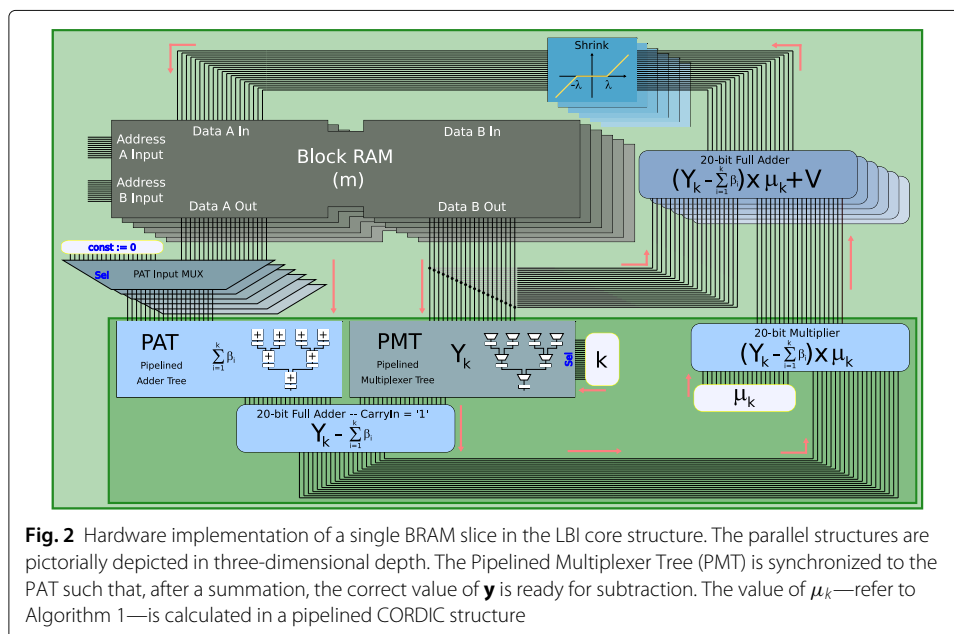

Fig. 2 Hardware implementation of a single BRAM slice in the LBI core structure. The parallel structures are pictorially depicted in three-dimensional depth. The Pipelined Multiplexer Tree (PMT) is synchronized to the PAT such that, after a summation, the correct value of **y** is ready for subtraction. The value of μ_k—refer to Algorithm 1—is calculated in a pipelined CORDIC structure

of $\boldsymbol{\beta}$ is subtracted from (refer to line 7 of Algorithm 1). The architecture of the PMT is such that the number of stages meets that of the PAT, so synchronization between the two outputs is naturally ensured. Furthermore, the selection key that acts on each stage of the PMT is derived from the cyclic iteration index, k.

The value of $\mu_k = \frac{1}{k}$, which involves a computation-heavy division, has been delegated to a pipelined CORDIC structure instead of a Look-Up Table (LUT) since the goal is to delegate the BRAMs for storage of vectors $\boldsymbol{\beta}$, $\mathbf{v}$, and $\mathbf{y}$. The stages of the pipeline are pre-filled before the iterations are started, requiring a number of clock cycles equal to its length for complete fill-up: this number has been chosen to be $F = 20$, providing an accurate estimation of the results of $1/k$. Moreover, stage propagation is enabled at each new iteration, ensuring that the correct value is always available without limiting the number of clock cycles per iteration. The reason behind choosing the pipelined CORDIC structure instead of a LUT for the $1/k$ calculation is, then, threefold: (i) preventing memory to be reserved for data other than the vectors $\boldsymbol{\beta}$, $\mathbf{v}$, and $\mathbf{y}$; (ii) no negative implications on the maximum clock frequency of the design, as evidenced by the Place and Route results of the full structure—refer to Section 5; and (iii) no negative influence on the number of clock cycles for each iteration since the initial F clock cycles are not a part of the iteration but, rather, of the initialization step.

Based on this memory structure, the amount of clock cycles necessary to complete the calculation of d (lines 5 to 8 in Algorithm 1) depends both on the number of data points and on the depth of the PAT, which, in turn, depends on the number of instantiated (or available) parallel BRAMs in the hardware structure. For an arbitrary iteration cycle, with cyclic index k, the equation that relates these values to the total number of clock cycles is $C_r' = \left\lceil \frac{k}{M} \right\rceil + \lceil \log_2 M \rceil$, where the subscript refers only to the reading and processing of $\boldsymbol{\beta}$ values up to the output of the PAT. Taking into account also the subsequent subtraction and multiplication steps—refer to Fig. 2—each taking one clock cycle, the total number of clock cycles amounts to $C_r = \hat{t} + \lceil \log_2 M \rceil + 2$, where $\hat{t} = \left\lceil \frac{k}{M} \right\rceil$ denotes the maximum value of t during an iteration.

4.2 PAT input control

Even though a *parallel row* is accessible at each clock cycle due to the parallel instantiation of the BRAMs, clearly not all values in the row will be used during a given iteration with index k. For that reason, a multiplexer (*PAT input MUX* in Fig. 2) is connected immediately after the BRAM output with its remaining input connected to a null value. Due to the additive identity property of zero, the output of the multiplexer can be directed to the PAT without the corruption of the result while accommodating the parallel storage structure.

The selection signal that controls the *PAT input MUX* is derived based on the fact that replacing BRAM outputs by zero is only necessary during the last *parallel row* access, i.e., when $t = \left\lceil \frac{k}{M} \right\rceil = \hat{t}$. Selection is, thus, based on an auxiliary counter that records the aforementioned value and on a so-called unary code (or thermometer code), which encodes the last column index that contributes to the sum. Figure 3 depicts the control unit responsible for handling the BRAM input address and selection of PAT inputs.

A NEW ITER strobe generated by the control unit and the clock signal are the necessary inputs. The cyclic iteration index counter k is implemented through a simple counter with parallel load dependent on the comparison with the signal length N. The unary code
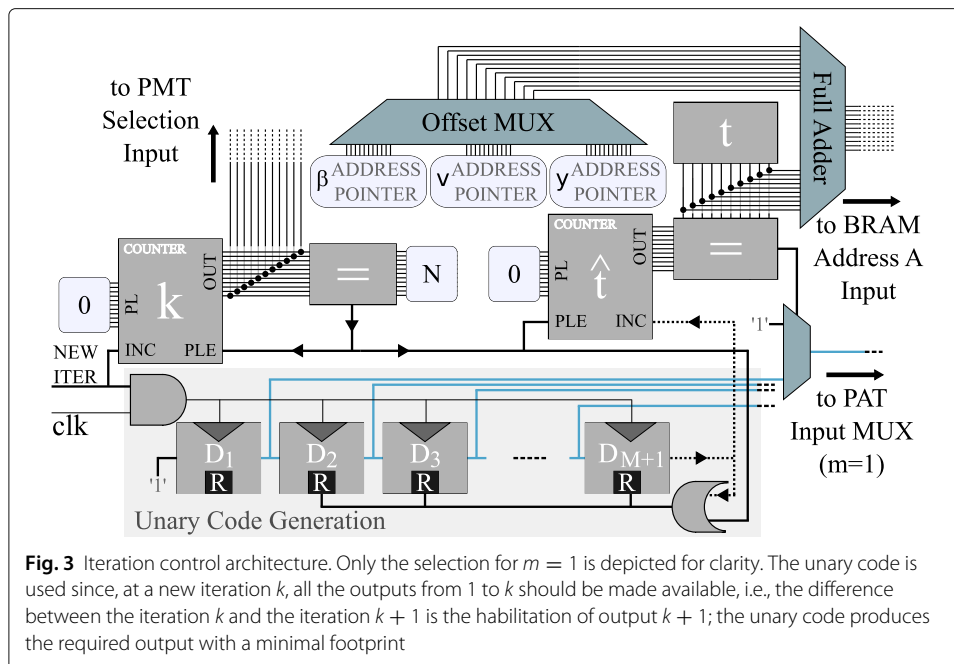
Fig. 3 Iteration control architecture. Only the selection for $m = 1$ is depicted for clarity. The unary code is used since, at a new iteration k, all the outputs from 1 to k should be made available, i.e., the difference between the iteration k and the iteration $k + 1$ is the habilitation of output $k + 1$; the unary code produces the required output with a minimal footprint

propagates at each new iteration, and when the $(M + 1)$ th stage is reached, it auto-resets while also incrementing the $\hat{t}$ counter. The unary code acts on *PAT input MUX* when $t = \hat{t}$ and the different address pointers are combined with the counter t to produce the correct BRAM address.

4.3 β and v storage

An indispensable step of the algorithm is the correct storage of the vectors β and $\mathbf{v}$ after processing. According to Algorithm 1, all the elements of vector β are processed by the shrink function right after processing of the vector $\mathbf{v}$. As previously pointed out, acceleration of the storage procedure tackles one of the algorithm's bottlenecks. Both the facts that the BRAMs allow for writing and reading from two independently addressed ports and that if λ is set to zero in the shrink function it implements the identity transformation have been harnessed to perform data storage optimization, as it is detailed as follows.

One of the BRAM's ports (taken as B without loss of generality in Fig. 2) is responsible for reading the values of $\mathbf{v}$ from the memory while the other port (A) is responsible for storing the values of β and $\mathbf{v}$. The addresses are controlled such that, on the first clock cycle, values of $\mathbf{v}$ in a *parallel row* are read (through port B), processed in the 20-bit full adder, and sent to the shrink function with $\lambda = 0$. Therefore, at the following clock cycle, the stable value of $\mathbf{v}$ can be stored (through port A) at the same time as the value of λ is changed in the shrink function and processes the values of $\mathbf{v}$ being read (through port B). In the third clock cycle, a stable value of β is stored (through port A) while the values of $\mathbf{v}$ from the following *parallel row* are accessed (through port B), initiating a new storage cycle for a subsequent *parallel row*.

The net number of clock cycles per *parallel row* storage is, thus, two if one does not compute the very first and last accesses; therefore, the number of clock cycles necessary at an arbitrary iteration with cyclic index k is $C_s = 2 \left\lceil \frac{k}{M} \right\rceil + 2 = 2\hat{t} + 2$. Two extra clock cycles are also necessary for the handshaking protocol between the iteration control unit

(presented in Fig. 3)—whose control over the BRAMs address is relieved—and the writing unit that takes over control and stores vectors β and v, i.e., $C'_s = 2\hat{t} + 4$.

The loading to and from the BRAM (either before starting the algorithm or after the analysis is performed) is performed via a Shift-Register that controls the Memory Enable of each of the BRAMs and routes the addresses and data ports to I/O ports of the entity. This way, data can be streamed and connected directly to all the input ports of the BRAMs while the addresses and Memory Enables ensure the correct loading of the values. Two extra clock cycles are necessary, one at the beginning of the streaming and one in the end, so the Shift-Registers are correctly initialized. Furthermore, to store and later access all three vectors, this streaming procedure must be repeated three times, sequentially, thereby creating the necessity of extra $2(2 + 3N)$ clock cycles for loading to and from the BRAMs. When compared to the number of clock cycles required by the iterations, however, it becomes clear that this number is negligible and has not been included in the clock cycle analysis at the end of the section. The data transfer hardware described above, which only corresponds to a small device utilization overhead, is included in the synthesis results of the next section.

4.4 Master FSM

In order to control all the aforementioned steps of the design, a so-called master Finite State Machine (FSM) is implemented. The states, transitions, and strobes depicted in Fig. 4 ensure the correct evolution of the algorithm. The FSM starts at an idle state and, based on an *init* strobe, evolves to the loading state, where control of the BRAMs is granted to the streaming structure. Once loading is done, the state evolves to the CORDIC initialization (for $1/k$ calculation) and, after that, to the iteration control state. This state initializes a counter, which saturates at the total value of iterations L; in case the counter value is below L, the state evolves first to the β summation (subsequently triggering a NEW ITER strobe—refer to Fig. 2) and, then, to the storage of β and **v**; if, however, the counter value is equal to L, the iterations are done, and unloading can start. Finally, after unloading is performed, the FSM returns to the idle state, where it waits for an upcoming *init* strobe.

4.5 Total clock cycle estimation

After analysis of the PAT processing and the data storage structure, the total number of clock cycles for an iteration can be determined. According to the previous analysis, combining C_s, the number of clock cycles necessary for storage with the determined C_r, the number of clock cycles necessary for the partial summation of β in the PAT, and the necessary operations to determine **y**, the total number of clock cycles spent in an arbitrary iteration with index k is $C_T = 3(\hat{t} + 2) + \lceil \log_2 M \rceil$.

The total number of clock cycles taken by the algorithm to elapse can be easily derived from this equation by summation over L, the total number of iterations:

$$C = F + \sum_{i=1}^{L} \left[3 \left(\left\lceil \frac{((i-1)\,\%N) + 1}{M} + 2 \right\rceil \right) + \lceil \log_2 M \rceil \right]. \tag{6}$$

The factor F in Eq. 6 accounts for pre- and post-processing instructions performed by the control unit such as master resets, granting control over the BRAMs, and, most importantly, preemptively filling up the pipelined CORDIC that calculates μ_k. However, as will
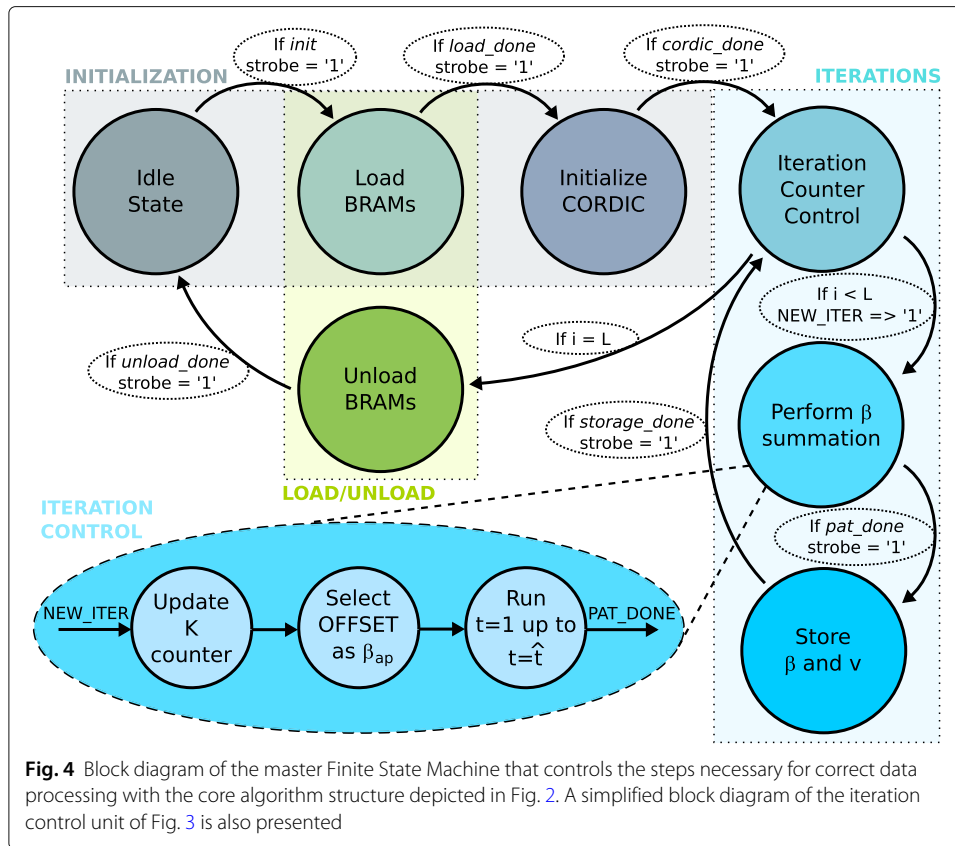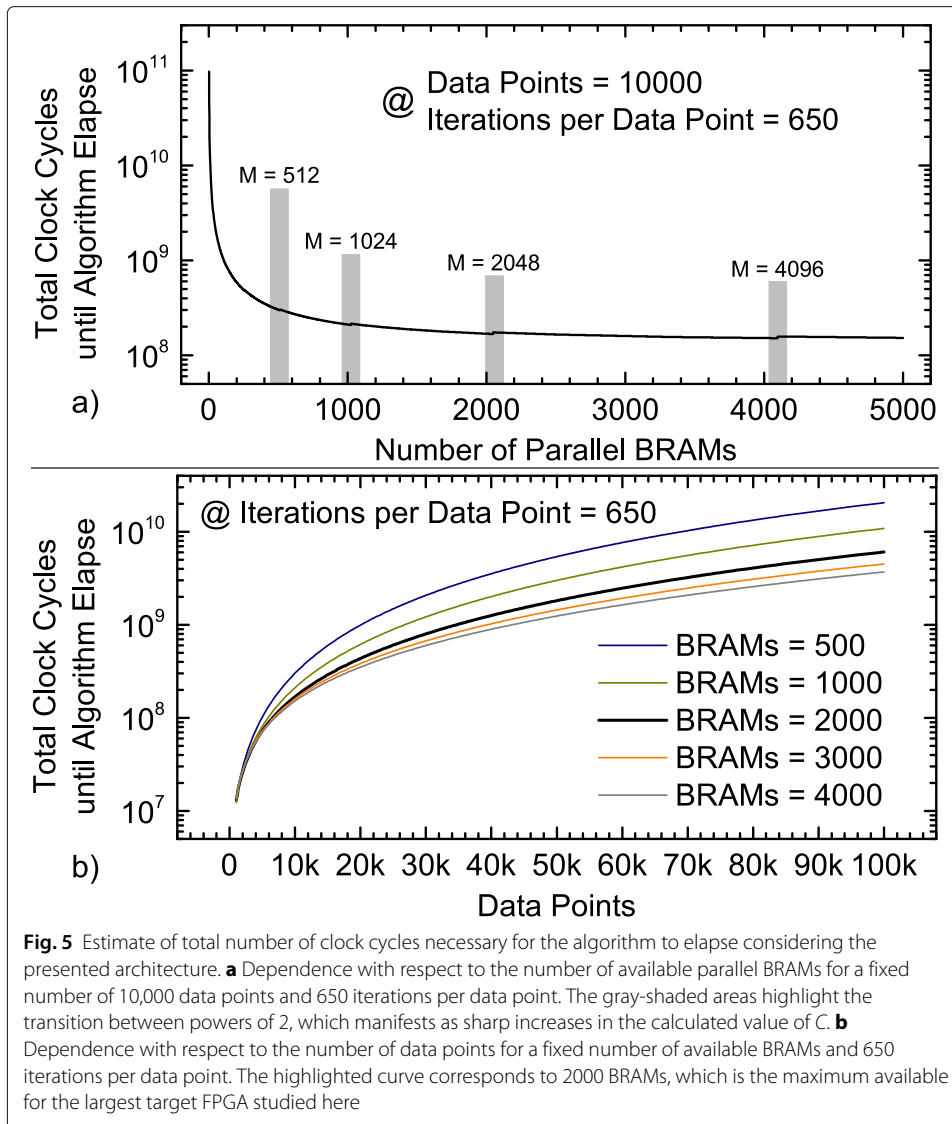
Fig. 4 Block diagram of the master Finite State Machine that controls the steps necessary for correct data processing with the core algorithm structure depicted in Fig. 2. A simplified block diagram of the iteration control unit of Fig. 3 is also presented

be described in the next section, the value of F is much smaller than the total number of clock cycles taken by the core procedure.

Figure 5 presents the dependence of the total number of clock cycles until the algorithm elapses with both the number of available parallel BRAMs for a fixed number of data points and with the number of data points for a fixed number of available BRAMs. In both cases, the iterations per data point (defined as L/N) are fixed at 650, a realistic value that will be discussed in further sections. Considering a maximum clock frequency achievable in the target FPGA to be around 100 MHz, a 10,000-point data series would be processed in less than 2 s, which represents an approximately 100 gain factor when compared to the Julia implementation reported in [1].

The results of Fig. 5a provide an interesting analysis point: a stagnation of the contribution of M, as it increases, to the decrease of the number of clock cycles necessary for the algorithm to elapse. As Eq. 6 indicates, the impact of M in the total number of clock cycles is of the form $1/M$. Therefore, as M increases, its contribution to decreasing C becomes relatively marginal. Moreover, it is important to note that there is also a second part in the term inside the summation of Eq. 6, which scales up with M, i.e., $\lceil \log_2 M \rceil$. Therefore, as the $1/M$ term reaches a relative stagnation with higher M, the contribution of the $\lceil \log_2 M \rceil$ term grows, which balances out for high values of M creating the observed stagnation, which is visually pronounced in Fig. 5 due to the logarithmic scale.

Fig. 5 Estimate of total number of clock cycles necessary for the algorithm to elapse considering the presented architecture. **a** Dependence with respect to the number of available parallel BRAMs for a fixed number of 10,000 data points and 650 iterations per data point. The gray-shaded areas highlight the transition between powers of 2, which manifests as sharp increases in the calculated value of *C*. **b** Dependence with respect to the number of data points for a fixed number of available BRAMs and 650 iterations per data point. The highlighted curve corresponds to 2000 BRAMs, which is the maximum available for the largest target FPGA studied here

5 Validation and synthesis

Comparison between the software-defined hardware implementation of the Linearized Bregman Iteration algorithm using the architecture presented in the previous section and its software implementation counterpart [1] permits validating the former. In order to provide a bit-true validation, the *SFIXED* standard used in the VHDL simulation was implemented in Julia allowing one to accompany, step-by-step, the evolution of the algorithm on both platforms and identify any discrepancies. Due to the fact that the rounding procedure is the same for both, no such discrepancies were observed; the fixed point Julia simulation code outputs *exactly* the same values as of the hardware implementation. The validation of the hardware implementation and the demonstration of its equivalence to the Julia *SFIXED* implementation create a versatile tool to estimate the performance of the FPGA results on a software environment.

For the simulation of the hardware implementation, the MODELSIM VHDL simulation environment was employed. In such an environment, both the evolution of the algorithm

as well as the number of clock cycles necessary to run each iteration can be extracted, so the results of Eq. 6 can also be ascertained. Even though an extremely reliable and versatile tool, VHDL simulation offers a drawback in terms of running time: simulating a high number of BRAMs or a large dataset can be extremely time-consuming. For this reason, a predetermined set of parameters (data points, iterations, and number of BRAMs) were chosen to showcase the validity of the hardware implementation.

Table 1 contains the information regarding the simulation of the hardware structure under the different parameter conditions, where B stands for the number of BRAMs, and L and N follow the previously defined notation. The estimated number of clock cycles based on Eq. 6 that appear in Table 1 takes into account the required $F = 21$ extra clock cycles for initialization and control, but excludes the up/down loading steps of data into and out of the BRAMs. The asterisk in the last column indicates that 2048 BRAMs are actually above the 2000 maximum available number of BRAMs with 20 bit-wide data entries in the target ALTERA STRATIX V FPGA, but could be implemented in a larger device.

The results of Table 1 are in excellent agreement with the expectations, which translate into the following: validation of the hardware implementation as well as a demonstration of its equivalence to the Julia *SFIXED* software implementation, and verification of the validity of Eq. 6, which, in turn, is a validation of the results of Fig. 5. The concluding step of this section is, then, to synthesize the hardware so that the maximum achievable clock frequency can be extracted. As previously commented, the clock frequency, combined with the total number of clock cycles necessary for the algorithm to elapse, can be used to estimate the amount of time the algorithm will take to execute. Furthermore, as a by-product of the synthesis results, it is possible to assess the percentage of FPGA resources occupied by the architecture, which, in turn, provides the means for selecting the target FPGA for hardware implementation. The results are summarized in Table 2, where the INTEL QUARTUS PRIME synthesis software was used. It is important to note that, due to the complexity of the Place and Route (PAR) problem, it is not reasonable to assume that the synthesis software will always find the global optimum. Therefore, the results of Table 2 should be interpreted as lower bounds of the optimal achievable clock frequency for each design instance and the discrepancies in these to be within the uncertainty of the PAR procedure.

Table 1 VHDL simulation–validation

	$N = 10$ $B = 4$ $L = 10$	$N = 10$ $B = 4$ $L = 100$	$N = 100$ $B = 4$ $L = 100$	$N = 100$ $B = 4$ $L = 1000$	$N = 1000$ $B = 4$ $L = 1000$
C (Eq. 6)	155	1361	4721	47021	384521
Clk. Cyc.	155	1361	4721	47021	384521

	$N = 1000$ $B = 128$ $L = 1000$	$N = 5000$ $B = 1024$ $L = 5000$	$N = 10000$ $B = 1024$ $L = 10000$	$N = 15000$ $B = 1024$ $L = 15000$	$N = 15000$ $B = 2048^*$ $L = 15000$
C (Eq. 6)	26269	124301	321781	592461	442989
Clk. Cyc.	26269	124301	321781	592461	442989

Table 2 Target FPGAs' synthesis and timing results. Processing times calculated for $N = 10,000$ and with $L = 6.5 \times 10^6$

| BRAMs | Stratix V: 5SGSMD5K2F40C2 | | | | | Cyclone V: 5CSXFC6D6F31C6 | | | | |
	ALMs out of 172,600	Registers	Memory [bits] out of 41,246,720	Max. Clk. Freq. [MHz]	Proc. time [s]	ALMs out of 41,910	Registers	Memory [bits] out of 5,662,720	Max. Clk. Freq. [MHz]	Proc. time [s]
1024	135,571 (79%)	88,938	20,971,520 (51%)	109.9	1.91	—[1]	—[1]	—[1]	—[1]	—[1]
512	68,135 (39%)	45,595	10,485,760 (25%)	166.97	1.78	—[1]	—[1]	—[1]	—[1]	—[1]
256	36,098 (21%)	25,147	5,242,880 (13%)	173.28	2.78	36,548 (87%)	24,741	2,621,440 (46%)	81.13	5.94
128	19,196 (11%)	13,839	2,621,440 (6%)	182.12	4.70	20,178 (48%)	13,475	1,310,720 (23%)	95.37	8.98
16	4982 (3%)	4012	327,680 (< 1%)	182.35	33.83	5000 (12%)	3928	163,840 (3%)	92.94	66.37
4	3435 (2%)	2925	81,920 (< 1%)	187.86	130.08	3427 (8%)	2929	40,960 (< 1%)	91.73	266.40

[1] Design too large to fit into device

Up to 1024 BRAMs could be instantiated in the STRATIX V, with as high as 109 MHz maximum clock frequency yielding a 1.91-s processing time for 10,000-long data series considering 650 iterations per sample (overall, 6.5 million iterations). This result represents a 100 speed-up factor in processing time with respect to software implementations under the same data conditions but running on a INTEL XEON CPU E5-2690 V4 at 2.6 GHz and 512 GB RAM [1], a major achievement, which advocates for the dedicated hardware solution applied to the trend break detection problem. It is also interesting to note that, for a smaller FPGA, the CYCLONE V, a $\sim$ 16 gain factor with respect to the software implementation was achieved, which is interesting in the sense that smaller FPGAs exhibit, generally, significantly lower costs, but could still deliver processing times in the range of a few seconds.

The results from Table 2 also indicate that a compromise between instantiation of a higher number of BRAMs (which reduces the total number of clock cycles necessary for the algorithm to elapse as determined by Eq. 6) and the maximum achievable clock frequency exists. In fact, the processing time for 512 instantiated BRAMs was lower than that of 1024 because the gain in clock frequency superseded that of the reduction of clock cycles. Again, it should be mentioned that the PAR problem is an extremely complex one and the algorithms that solve it may not always reach the best possible solution, so the clock frequency values obtained should be interpreted as lower bounds. Finally, to put the results into an application proned perspective, fiber profiles as long as 50 km could be analyzed in search for breaks in under 10 s [1].

6　Results and discussion

6.1　Case study results

Validation of the software Julia *SFIXED* implementation performed in Section 5 allows one to investigate aspects of the hardware implementation in a more suitable simulation environment. This is important due to the amount of simulation workload necessary to yield statistically relevant results. To provide a complete overview, the analysis is split into two steps. First, an experimental dataset extracted in a laboratory environment, with standard measurement devices, is processed by the bit-true validated Julia *SFIXED* implementation and by the original 64-bit floating version of the algorithm [1]. These results are evaluated in terms of their trend break detection capabilities or, in other words, the performance of the two versions of the algorithm.

After analysis with real-world data, which, unfortunately, is limited to the availability of resources in the laboratory, the second step is to submit the algorithm to simulated datasets that contain the same features as the real-world ones; in [1], the creation of a testbench of simulated datasets is discussed in detail, specifically in regard to noise addition, with very pronounced resemblance between the real-world and simulated results; this, in turn, allows for statistically relevant investigation of the performance of the algorithm. It should be noted that, for all the results presented in this section, the bit-width of data points for the *SFIXED* format was fixed at 20, where the reason behind this will be clarified in Section 6.1.2.

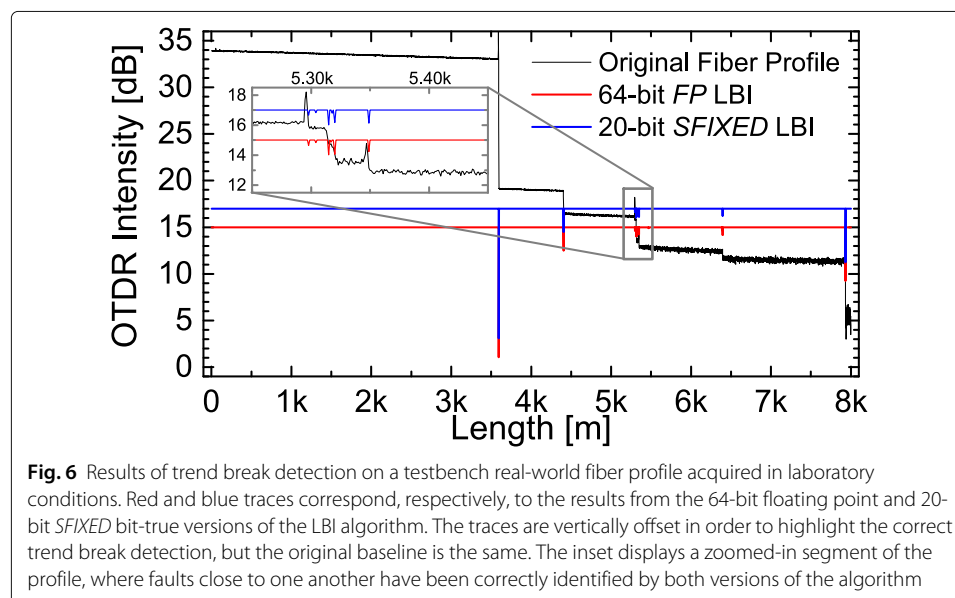#### 6.1.1　Analysis of a real-world dataset

The subject of the experimental analysis of the proposed hardware implementation was chosen to be the dataset corresponding to the measurement of an optical fiber profile.

The data acquisition was performed with a so-called Optical Time-Domain Reflectometry (OTDR) device based on an FPGA [10]. The OTDR provides the user with information of the fiber's integrity (the fiber profile) by measuring the optical power that is reflected back from the optical fiber when a probing pulse is launched into it [26]; the result is displayed in logarithmic scale (dB), as depicted by the black trace in Fig. 6. This choice goes along with the motivation of Section 1, since the acquisition system and the proposed hardware implementation of the analysis algorithm could, potentially, coexist in the same FPGA. The chosen fiber profile exhibits several interesting features related to fault detection: the presence of both high magnitude and low magnitude faults and faults separated by few points.

The fact that the LBI algorithm, in its original version, provides accurate trend break detection results had already been determined in [1] and is again verified by the red trace in Fig. 6. The striking feature showcased in Fig. 6, however, is the fact that the bit-true validated *SFIXED* version of the algorithm, the blue trace, exhibits equivalent results, indicating that the proposed hardware implementation of the LBI algorithm using the 20-bit *SFIXED* format upholds the performance of its 64-bit floating point counterpart. One important comment is regarding the linear slope component that is clearly part of the original signal in Fig. 6. In [1], the slope component has been included in the candidate matrix **A** for completeness; however, this slope can be pre-compensated since it is a standard value for most commercial optical fibers as specified in ITU-T G.652. In order to maintain the LBI algorithm, as presented in Algorithm 1, general with respect to any trend break detection problem but, at the same time, allow for data from optical fiber monitoring to be processed by this version of the algorithm, the slope has been pre-compensated.

6.1.2   Testbench of simulated datasets

The objective of using a set of simulated datasets, and, therefore, accumulating statistically relevant data (results of over 15,000 different simulated datasets were analyzed),

Fig. 6 Results of trend break detection on a testbench real-world fiber profile acquired in laboratory conditions. Red and blue traces correspond, respectively, to the results from the 64-bit floating point and 20-bit *SFIXED* bit-true versions of the LBI algorithm. The traces are vertically offset in order to highlight the correct trend break detection, but the original baseline is the same. The inset displays a zoomed-in segment of the profile, where faults close to one another have been correctly identified by both versions of the algorithm
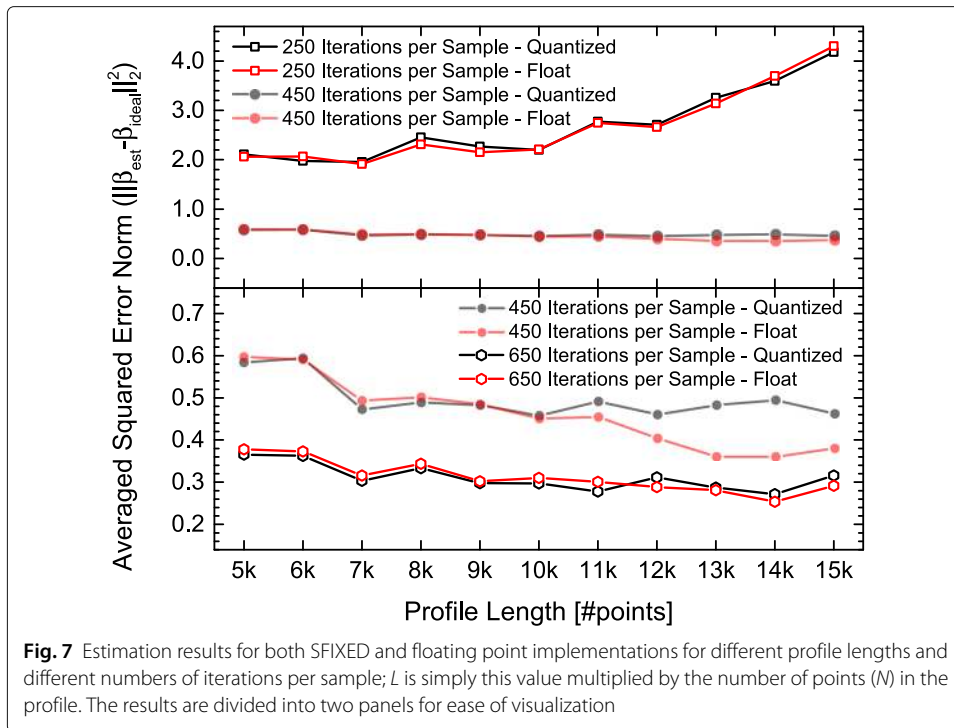
is twofold: firstly, as it was mentioned in Section 3, to empirically determine the interval of number of iterations per sample for which the estimation quality reaches a reliable level and, secondly, to compare the quality of estimation between the 64-bit floating point implementation, and the 20-bit *SFIXED* implementation for profiles with different number of data points and with different number of iterations per sample.

Evaluation of the bit-width parameter has been conducted based on the limited number of possible bit-width configurations of the BRAMs in the target FPGAs considered here; supported bit-widths for both the STRATIX V and the CYCLONE V are 10, 20, and 40 bits [27]. All three configurations have been tested, and the results are as follows. The 10-bit *SFIXED* implementation proved to be too limited in terms of correctly expressing the input data, since large discrepancies with respect to the benchmark results ran in a personal computer with a 64-bit floating point precision have been observed. As the bit-width was increased to 20, these discrepancies were drastically diminished, indicating a more equivalent expression of the data series and, also, a sufficient arithmetical dynamic range. Further increasing the bit-width to 40, however, did not effect any significant change in the performance of the algorithm, and since this configuration requires twice the memory usage when compared to the 20-bit one, while also increasing the complexity of the arithmetical structures in the FPGA, the value of 20 bits has been set as an operational parameter for all the tests.

To clarify this analysis, an estimation error metric has been used, following the definition in [1]: noiseless sparse vectors with randomly sorted magnitudes ($\boldsymbol{\beta}_{\text{ideal}}$) are used to create datasets using the candidate matrix $\mathbf{A}$, to which white Gaussian noise is added; the results of the estimation are $\boldsymbol{\beta}_{\text{est}}$, which are then compared to $\boldsymbol{\beta}_{\text{ideal}}$ using the squared error norm. The closer to zero error between the estimated and ideal vectors, the better is the estimation. The results of such estimation error metric are depicted in Fig. 7 for different values of L and for both the 20-bit *SFIXED* and the 64-bit floating point implementations; the results are extremely similar, indicating that no information is lost due to the 20-bit *SFIXED* implementation. Here, the intrinsic slope of optical fiber profiles has not been considered once again, as discussed in the previous sub-section.

From the results of Fig. 7, it is possible to conclude that the differences in estimation between the 20-bit *SFIXED* and 64-bit double implementations are negligible, i.e., the hardware implementation will have no problems achieving comparable estimation accuracy as the software version, e.g., in [1]. A large variety of test cases (different lengths and fault scenarios) have also been tested, and the results point to the fact that a 20-bit fixed point resolution provides comparable results in terms of detection performance when compared to longer bit-length words. The conclusion, therefore, is that significant deviations for other datasets are highly unlikely. Furthermore, it is important to note that, as discussed in depth in [1], it is impractical to expect the fault detection algorithm to be able to resolve any fault (with an arbitrary magnitude) for two main reasons: first, the amount of information gained from identifying a fault below a certain level (say, smaller than 0.1) is close to none, since the impact of this fault on the optical link transmission is minimum; second, in order to achieve a level of sensitivity that allows one to identify extremely small faults would require the processing time to be also extremely high, as also discussed in [1]. With that being said, it can be expected, from the results, that the gain in sensitivity from increasing the bit-length of the words beyond 20 falls within a practical sensitivity issue and, thus, becomes irrelevant.

Fig. 7 Estimation results for both SFIXED and floating point implementations for different profile lengths and different numbers of iterations per sample; *L* is simply this value multiplied by the number of points (*N*) in the profile. The results are divided into two panels for ease of visualization

Furthermore, it becomes clear that, after 450 iterations per sample, the accuracy stabilizes, with an averaged squared error norm value in the order of 0.5. This indicates that the results from Fig. 5 with 650 iterations per sample are indeed realistic. Finally, this result is also useful when interpreted along with those of Fig. 5: a compromise between the total number of clock cycles before the algorithm elapses and the quality of the estimation can be found, and in specific cases, one of these can be sacrificed (increasing the processing time or allowing for a worse estimate) to boost the other (faster results or extremely precise estimation). In real-world applications, the expected accuracy (given a set of parameters including bit-length and total number of algorithm iterations) can be determined in advance and provided to the user, so that the constraints are known a priori. If necessary, the memory word length can be adjusted such that it is larger or equal to the bit-width that yields the desired calculation precision.

6.2 State-of-the-art hardware implementations of sparse estimation algorithms—comparison with the proposed architecture

The LBI algorithm, in order to be suitable for tackling the trend break detection problem, was modified from its original format; the especial structure of the candidate matrix, for instance, allowed one to derive the expressions in Eqs. 3 and 4, which greatly simplify the procedure [1]. This fact prompts the natural question of whether there are other methodologies that, even though originally intended to be used in sparse estimation, could be steered towards the application at hand. Moreover, this would allow for a comparison between the herewith proposed hardware implementation of the LBI algorithm with other similar structures. An extensive literature review on this subject reveals that, when it comes to hardware implementations of sparse estimation methods, the majority

of the works focus on the Orthogonal Matching Pursuit (OMP) algorithm [28], which has, thus, been the focus of this comparison.

Although an interesting approach, where efficient architectures have been proposed such as [29, 30], OMP typically assumes that the number of non-zero entries of the vector $\boldsymbol{\beta}$—and, thus, the number of candidates that compose the signal of interest—is known. For applications where this assumption is valid, the method is highly efficient and, in addition, provides recovery guarantees, i.e., the certainty that, if present, the candidate will be found. For the cases that have been described in the introductory section of this document, however, this is a rather detrimental characteristic; for the specific case of trend break detection associated to fiber fault detection, for instance, this assumption would yield results that do not translate the true underlying trend of the original signal. Furthermore, the OMP algorithm relies, for optimal performance, on a small value of the so-called *mutual incoherence*:

$$\mu = \max_{i \neq j} \frac{|\tilde{\mathbf{a}}_i \tilde{\mathbf{a}}_j|}{\|\tilde{\mathbf{a}}_i\|_2 \|\tilde{\mathbf{a}}_j\|_2}, \tag{7}$$

with $\tilde{\mathbf{a}}_i$, $\tilde{\mathbf{a}}_j$ as the ith and jth columns of $\mathbf{A}$, respectively; OMP requires the value μ to be smaller than $\frac{1}{2k-1}$ for guaranteed recovery of the sparse positions [31]. As one can easily check from (2), in the case of the candidate matrix associated to trend break detection, the value μ is close to its maximum value of 1 when it comes to close neighboring columns of $\mathbf{A}$, which could significantly decrease the OMP algorithm's performance in this scenario.

In order to propitiate an interesting comparison and further discussion about the disparities between the two algorithms and, moreover, between their respective hardware implementations, a framework for utilizing the OMP algorithm in a trend break detection scenario has been developed and is as follows. A sparsity factor ξ is defined, and the maximum number of non-zero elements within the estimation problem is determined using ξ and the total number of points in the dataset of interest, N; this maximum non-zero number is herewith defined as $N_0 = \xi N$. It is important to note that, since the actual number of non-zero elements that are present in the signal of interest is not known a priori, this approach will always be suboptimal when compared to an algorithm, such as the LBI, that does not necessitate such information. The OMP algorithm is, then, ran normally with N_0 iteration loops while the intermediate estimation results are submitted to a consistent information criterion for determining the best sparse approximation; the Bayesian Information Criterion [32], which balances the squared error norm and the resulting sparsity for model identification, has been employed. The complete procedure, dubbed OMP FOR TREND BREAK DETECTION, is structured in Algorithm 2 and has been directly adapted from [33].

The OMP FOR TREND BREAK DETECTION algorithm involves three main procedures, defined in Algorithm 2, as follows. The optimization problem of line 7 allows one to find the candidate with the highest overlap with the residual of previous iterations. This candidate is removed from the measured signal, and the new residual is determined (in line 11) with an intermediate estimate ($\hat{\boldsymbol{\beta}}_i$) being generated as a by-product (in line 10). Finally, the best intermediate estimate is found using the BIC and output as the final OMP estimate.

6.2.1 Performance comparison based on software implementation

Being a well-known algorithm in the sparse estimation community, several implementations of the OMP can be found as well as detailed explanations of the procedure; here, a

---

Algorithm 2 OMP for Trend Break Detection

Require: Measurement vector **y**, Sparsity Level ξ,

Candidate Matrix **A**

Ensure: Estimated $\hat{\boldsymbol{\beta}}_{\text{out}}$

1: $\mathbf{r}_0 = \mathbf{y}$
2: $\boldsymbol{\Phi}_0 = [\ ]$
3: $\Lambda_0 = \{\emptyset\}$
4: $N_0 = \xi N$
5: $\text{BIC}_{\text{res}} = \infty$
   $\rightarrow$ Innitiate OMP algorithm
6: **for** $i = 1 : N_0$ **do**
7: $\quad \lambda_i = \arg\max_{j=[1,N]/\Lambda_{i-1}} |\langle \mathbf{r}_{i-1}, \mathbf{A}_j \rangle|$
8: $\quad \Lambda_i = \Lambda_{i-1} \bigcup \{\lambda_i\}$
9: $\quad \boldsymbol{\Phi}_i = [\boldsymbol{\Phi}_{i-1} \, \mathbf{A}_j]$
10: $\quad \hat{\boldsymbol{\beta}}_i = \arg\min_x ||\boldsymbol{y} - \boldsymbol{\Phi}_i x||$
11: $\quad \mathbf{r}_i = \mathbf{y} - \boldsymbol{\Phi}_i \hat{\boldsymbol{\beta}}_i$
   $\rightarrow$ Introducing the BIC as a function
12: $\quad \text{BIC}_{\text{new}} = \text{BIC}\left(\hat{\boldsymbol{\beta}}_i, i\right)$
13: $\quad$ **if** $\text{BIC}_{\text{new}} < \text{BIC}_{\text{res}}$ **then**
14: $\quad\quad \hat{\boldsymbol{\beta}}_{\text{out}} = \hat{\boldsymbol{\beta}}_i$
15: $\quad\quad \text{BIC}_{\text{res}} = \text{BIC}_{\text{new}}$
16: $\quad$ **end if**
17: **end for**

Cholesky-decomposition-based OMP has been used as reference [33]. In order to maintain consistency with respect to the Julia implementation of the LBI, the code has been written in Julia.

The first comparison step, based on the software implementation, allows for the comparison of both algorithms to be analyzed. This entails, as set forth in [1], the evaluation of figures of merit such as the *precision* and the *Matthews correlation coefficient* (MCC) [34], where a so-called contingency table is used to derive the aforementioned figures of merit, and revolves around the determination of so-called true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) associated with the detection of breaks in the dataset. Assessing the performance with statistically relevant results is an important part of the performance comparison, so the framework of the testbench of simulated datasets, discussed in Section 6.1 B, has been once again employed. For the results herewith presented, the maximum number of trend breaks included in the simulated datasets was five, and since the comparison is not related to timing, ξ was set as 0.01, i.e., 1% of the total number of points in the dataset. Finally, N was varied from 5000 to 15,000 so that different dataset sizes could be considered in the analysis.

As the results of Table 3 demonstrate, the performance of the OMP FOR TREND BREAK DETECTION does not reach the performance level exhibited by the LBI, even when the sparsity level N_0 is made much larger than the actual number of breaks present in the dataset. On the one hand, the precision indicates how confident one can be about a break identified by the algorithm, which represents a huge impact on the application. If one considers, for instance, an optical fiber network manager that must take actions regarding the repairment of a cable, the cost of deploying a mobile unit creates a demand for high precision in the analysis results. On the other hand, the MCC is an interesting figure of merit from a theoretical point of view, since it stands for a correlation coefficient between

Table 3 Performance comparison

Algorithm	Precision $\left(\frac{TP}{TP+FP}\right)$	MCC $\left(\frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}\right)$
LBI	67.3%	0.81
OMP	8.2%	0.28

the observed and predicted binary classifications; in this case, the break/no-break classification: a value of $+1$ corresponds to perfect prediction and a value of 0 corresponds to random classification results. It is possible to observe, with this respect, that the LBI algorithm is very much closer to correctly predicting the underlying trend of the dataset (with an MCC of 0.81) than the OMP algorithm, which exhibits a low MCC value of 0.28. The conclusion of this analysis indicates that, even in case the performance of the OMP is mantained when the structure is adapted to hardware, as has been demonstrated in Section 5 A and B for the LBI, it will remain as a suboptimal choice for trend break detection with respect to its performance.

6.2.2 Comparison based on hardware implementation

The in-depth description and analysis of the LBI hardware structure presented in Section 4 allowed one to derive the total number of clock cycles taken, for a given set of input parameters, for the algorithm to elapse. Similar analysis for the OMP is, unfortunately, not available in the literature; since the focus of this document is not on the hardware implementation of the OMP, the comparison must, then, be performed using a different figure of merit. In [33], two interesting figures of merit are presented: the total processing time (0.34 ms) for fixed parameters (a degree of sparsity of 36 (p_1), a (p_2) 256-long measurement vector and 1024 (p_3) samples), and the total number of multiplication and addition operations as a function of $p_{1..3}$. There, the data precision, in bits, was 18, as opposed to the 20-bit *SFIXED* considered, here, for the LBI, which is within a comparable range.

For the closest comparable instance size for trend break detection (1024×1024), the hardware implementation of the LBI would elapse (considering the data in Table 2 in 64 ms, an ∼200 factor with respect to the OMP). This result is, however, ambiguous and further analysis is necessary, which has been subdivided into complexity and device usage. Analysis of the complexity has been carried out indirectly by evaluating the total number of multiplications necessary for both hardware architectures. The total number of multiplications necessary for the OMP increases cubically [33] in the degree of sparsity and quadratically in the instance size, while that for the LBI increases linearly (as discussed throughout Sections 3 and 4). Even though for small instance sizes, such as the previously mentioned 1024×1024 problem, the OMP exhibits a total number of necessary multiplications much lower than the LBI, the relation quickly inverts as the instance size is increased to, for instance, 15,000×15,000, a quite small dataset for optical fiber analysis, and higher. This result clearly evidenciates why, in [33], the digital signal processing (DSP) core of the target FPGA is almost at its usage limit for a relative small instance size and the fact that, as the instance size increases, the OMP becomes a less attractive solution to trend break detection.

Even though the target FPGAs for the architecture presented here (the ALTERA STRATIX V) and in [33] (the XILINX VIRTEX-6) are not the same, they are in the same

range of FPGA sizes and, thus, allow for device usage comparison. For the OMP, the device usage (where memory is included) is associated to the necessary matrix inversion for line 10 in Algorithm 2 and the storage of the candidate matrix elements. Therefore, the size of the hardware depends on the problem size and increases tremendously if the degree of sparsity increases; in [33], the modest $p_1 = 36$, $p_2 = 256$, and $p_3 = 1024$ problem occupied 76% of the DSP slices available, 69% of the available BRAMs, and a total of 16% slices. In the proposed LBI architecture, not only the number of occupied BRAMs is more flexible (the architecture of the $\boldsymbol{\beta}$ matrix in Eq. 5 can be made more vertical or more horizontal) but also the relative number of occupied slices is also smaller; a 128-BRAM design, which enables up to 35,000-long datasets, for instance, would only occupy 11% of the FPGA, and no DSP slices, which, in this case, could be reserved for other useful procedures, such as the pre-processing mentioned in Section 3. It is interesting to mention that, according to Eq. 6, there is a trade-off between the number of used BRAMs and the total number of clock cycles necessary for the algorithm to elapse, which would play a role in case the number of BRAMs was chosen to be either smaller or greater. In both cases, however, it is clear that the overall device usage of the proposed LBI structure is much more feasible, flexible, and economic for datasets with $\geq$ 10,000 points.

7   Conclusions

Trend break detection, or level-shift detection, is a problem that permeates several science fields, and an efficient, accurate, and highly reliable processing unit to solve it is desirable. Combining the flexible hardware design tools of Field Programmable Gate Arrays and the efficient Linearized Bregman Iteration algorithm allowed for the development of such a unit. The manipulation of the data storage structure as well as the algorithm flow and control in hardware yielded an up to 100 times gain in processing time when compared to a personal computer while maintaining all the observed qualities of the algorithm, such as low estimation error and high level-shift detection precision. The speed-up factor greatly depends on the memory availability in the target hardware, and even though this specific speed-up factor has been calculated for the Stratix V FPGA, a chip with more than 1000 BRAMs (each containing 1024 20-bit-long words) can achieve the same speed-up level.

Due to its flexible memory structure, the proposed hardware architecture can be implemented in different-sized FPGAs, with the main distinctions being the amount of available dual-block RAMs and maximum achievable clock frequency, characteristics that are hardware-dependent. On a middle-sized chip such as the ALTERA CYCLONE V, the hardware supports up to 256 parallel BRAMs with a maximum clock frequency of 81 MHz and a total processing time of 6 s for a 10,000-long dataset and 650 iterations per sample. Such processing prowess can be directed towards on-line data supervision such as optical fiber monitoring, which constitutes an exciting future point of investigation. Furthermore, incorporating advanced signal processing techniques into the hardware design in order to eliminate any pre-processing step while increasing the convergence speed is also a sought-after goal for future studies.

Evaluation of the architecture using both real-world and simulated datasets making use of its bit-true hardware-validated software showed that the performance is not deteriorated due to necessary adjustments for hardware implementation. In fact, the comparison of the floating point and quantized versions of the algorithm yielded negligible discre-

pancies when analyzing the squared error norm of the estimation. Further comparative analysis of the performance and hardware architecture with respect to a state-of-the-art algorithm for sparse estimation showed that the performance of the LBI for trend break detection is much more pronounced; at the same time, even though slower for small instance sizes, the complexity of the LBI structure allows for manipulation of much longer datasets, which is a necessity for trend break detection in the context of optical fiber analysis.

In summary, the FPGA implementation of the Linearized Bregman Iteration algorithm adapted for trend break detection, reported in the present manuscript, has fomented the following contributions: the description of the hardware structure of the algorithm and an efficient parallel memory access structure; the results that such an implementation provides, with impacting gains in processing time without loss of performance; its comparison with state-of-the-art hardware implementations of sparse estimation algorithms (in particular, the Orthogonal Matching Pursuit algorithm), indicating clear advantages of the proposed architecture in terms of performance and hardware flexibility; and the future points of investigation that it enables in the field of digital signal processing in an FPGA, especially with respect to creating an embedded unit for data acquisition and processing with direct applications in optical fiber analysis.

Abbreviations

AGD: Approximate gradient descent; BIC: Bayesian Information Criterion; BRAM: Block RAM; CORDIC: COordinate rotation DIgital computer; CPU: Central processsing unit; DSP: DIgital signal processing; FN: False negatives; FP: False positives; FPGA: Field Programmable Gate Array; FSM: Finite State Machine; LASSO: Least absolute shrinkage and selection operator; LBI: Linearized Bregman Iterations; LUT: Look-Up Table; MCC: Matthews correlation coefficient; MUX: Multiplexer; OLS: Ordinary least square; OMP: Orthogonal Matching Pursuit; OTDR: Optical Time-Domain Reflectometry; PAR: Place and Route; PAT: Parallel adder tree; PMT: Parallel multiplexer tree; RAM: Random access memory; SFIXED: Signed fixed point; SOC: System On Chip; TN: True negatives; TP: True positives; VHDL: Very-high-speed integrated circuit hardware description language.

Authors' contributions

All authors contributed equally to this work, and read and approved the final manuscript.

Availability of data and materials

Data and source code are available from the corresponding author upon request.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]Center for Telecommunications Studies, Pontifical Catholic University of Rio de Janeiro, Marquês de São Vicente, Rio de Janeiro, 22451-900, Brazil. [2]QC2DLab, Kavli Foundation, Technical University of Delft, Mekelweg 5, 2628 CC, Delft, The Netherlands. [3]Institute of Signal Processing, Johannes Kepler University, Altenbergerstraße 69, 4040, Linz, Austria.

References

1. M. Lunglmayr, G. C. Amaral, Linearized Bregman iterations for automatic optical fiber fault analysis. IEEE Trans. Instrum. Meas. **68**(10), 3699–3711 (2018)
2. M. Basseville, A. Benveniste, Design and comparative study of some sequential jump detection algorithms for digital signals. Acoust. Speech Sig. Process IEEE Trans. **31**(3), 521–535 (1983)
3. L. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE. **77**(2), 257–286 (1989)
4. D. A. Lorenz, S. Wenger, F. Schöpfer, M. Magnor, *A sparse Kaczmarz solver and a linearized Bregman method for online compressed sensing. IEEE International Conference On Image Processing (ICIP)*, (2014), pp. 1347–1351
5. W. S. Rea, M. Reale, C. Cappelli, J. A. Brown, Identification of changes in mean with regression trees: an application to market research. Econ. Rev. **29**(5-6), 754–777 (2010)

6. M. Storath, A. Weinmann, L. Demaret, Jump-sparse and sparse recovery using potts functionals. IEEE Trans. Sig. Process. **62**(14), 3654–3666 (2014)

7. J. P. von der Weid, M. H. Souto, J. D. Garcia, G. C. Amaral, Adaptive filter for automatic identification of multiple faults in a noisy otdr profile. J. Light. Technol. **34**(14), 3418–3424 (2016)

8. S.-J. Kim, K. Koh, S. Boyd, D. Gorinevsky, ℓ_1 trend filtering. SIAM review. **51**(2), 339–360 (2009)

9. M. Lunglmayr, M. Huemer, in *Sensor Array and Multichannel Signal Processing Workshop (SAM), 2016 IEEE*, Efficient linearized Bregman iteration for sparse adaptive filters and Kaczmarz solvers (IEEE, 2016), pp. 1–5

10. F. Calliari, L. E. Y. Herrera, J. P. von der Weid, G. C. Amaral, in *Proceedings of the 6th International Conference on Photonics, Optics and Laser Technology - Volume 1: PHOTOPTICS,* High-dynamic and high-resolution automatic photon counting OTDR for optical fiber network monitoring. (INSTICC, 2018), pp. 82–90

11. K. J. Holyoak, Parallel distributed processing: explorations in the microstructure of cognition. Science. **236**, 992–997 (1987)

12. J. Xie, P. K. Meher, M. Sun, Y. Li, B. Zeng, Z.-H. Mao, Efficient fpga implementation of low-complexity systolic Karatsuba multiplier over gf (2m) based on nist polynomials. IEEE Trans. Circ. Syst. I: Regular Papers. **64**(7), 1815–1825 (2017)

13. P. Greisen, M. Runo, P. Guillet, S. Heinzle, A. Smolic, H. Kaeslin, M. Gross, Evaluation and fpga implementation of sparse linear solvers for video processing applications. IEEE Trans. Circ. Syst. Video Technol. **23**(8), 1402–1407 (2013)

14. L. Cong, W. Xiaofu, Design and realization of an fpga-based generator for chaotic frequency hopping sequences. IEEE Trans. Circ. Syst. I: Fundam. Theory Appl. **48**(5), 521–532 (2001)

15. (W. Vanderbauwhede, K. Benkrid, eds.), *High-Performance Computing Using FPGAs*. (Springer, New York, NY, 2013). https://doi.org/10.1007/978-1-4614-1791-0

16. J. Hu, W. Guo, J. Wei, R. C. C. Cheung, Fast and generic inversion architectures over$GF(2^m)$using modified Itoh-Tsujii algorithms. IEEE Trans. Circ. Syst. II: Express Briefs. **62**(4), 367–371 (2015). https://doi.org/10.1109/TCSII.2014.2387612

17. F. Cardells-Tormo, P.-. Molinet, Area-efficient 2-d shift-variant convolvers for fpga-based digital image processing. IEEE Trans. Circ. Syst. II: Express Briefs. **53**(2), 105–109 (2006). https://doi.org/10.1109/TCSII.2005.857091

18. C. Zhang, Z. Fang, P. Zhou, P. Pan, J. Cong, in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks, (2016), pp. 1–8. https://doi.org/10.1145/2966986.2967011

19. T. Hastie, R. Tibshirani, M. Wainwright, *Statistical Learning with Sparsity: the Lasso and Generalizations*, 1st edn. (CRC press, Monographs on Statistics and Applied Probability 143, 2015)

20. J.-f. Cai, S. Osher, Z. Shen, Fast linearized Bregman iteration for compressed sensing. Math. Comput. **78**(267), 1515–1536 (2008)

21. U. Meyer-Baese, Digital signal processing with field programmable gate arrays. Vol. 65 (Springer, Berlin, 2007)

22. F. Sato, K. Tsuchiya, Y. Nagao, T. Hirama, R. Oka, K. Takahashi, Ultra-high-fiber-count optical cable for data center applications. SEI Technical Review. **86**, 45–50 (2018)

23. A. Hornsteiner, Fiber optic technology trends in data transmission: digitalization of data advance the need for constant upgrading of data networks. Optik & Photonik. **12**(4), 20–24 (2017)

24. M. Lunglmayr, B. Hiptmair, M. Huemer, in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, Scaled linearized bregman iterations for fixed point implementation, (2017), pp. 1–4. https://doi.org/10.1109/ISCAS.2017.8050534

25. H. H. Bauschke, R. S. Burachik, P. L. Combettes, V. Elser, D. R. Luke, H. Wolkowicz, *Fixed-point algorithms for inverse problems in science and engineering*, vol. 49 (Springer Science & Business Media, 2011)

26. M. Barnoski, M. Rourke, S. Jensen, R. Melville, Optical time domain reflectometer. Applied optics. **16**(9), 2375–2379 (1977)

27. IntelFPGA, High-performance computing using FPGAs (2015). https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf

28. E. J. Candes, M. B. Wakin, An introduction to compressive sampling. IEEE Signal Proc. Mag. **25**(2), 21–30 (2008). https://doi.org/10.1109/MSP.2007.914731

29. X. Ge, F. Yang, H. Zhu, X. Zeng, D. Zhou, An efficient fpga implementation of orthogonal matching pursuit with square-root-free qr decomposition. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **27**(3), 611–623 (2019). https://doi.org/10.1109/TVLSI.2018.2879884

30. H. Rabah, A. Amira, B. K. Mohanty, S. Almaadeed, P. K. Meher, Fpga implementation of orthogonal matching pursuit for compressive sensing reconstruction. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **23**(10), 2209–2220 (2015). https://doi.org/10.1109/TVLSI.2014.2358716

31. T. T. Cai, L. Wang, Orthogonal matching pursuit for sparse signal recovery with noise. IEEE Trans. Inf. Theory. **57**(7), 4680–4688 (2011)

32. G. Schwarz, et al., Estimating the dimension of a model. Ann. Stat. **6**(2), 461–464 (1978)

33. H. Rabah, A. Amira, B. K. Mohanty, S. Almaadeed, P. K. Meher, Fpga implementation of orthogonal matching pursuit for compressive sensing reconstruction. IEEE Trans. very large scale Integr. (VLSI) Syst. **23**(10), 2209–2220 (2014)

34. B. W. Matthews, Comparison of the predicted and observed secondary structure of t4 phage lysozyme. Biochim. Biophys. Acta Protein Struct. **405**(2), 442–451 (1975)

Publisher's Note