Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values

Fabian Gieseke[1] Sabina Rosca[2] Troels Henriksen[1] Jan Verbesselt[2] Cosmin E. Oancea[1]

[1]*Department of Computer Science*
University of Copenhagen
Copenhagen, Denmark

[2]*GRS Laboratory*
Wageningen University
Wageningen, The Netherlands

***Abstract*—Large amounts of satellite data are now becoming available, which, in combination with appropriate change detection methods, offer the opportunity to derive accurate information on timing and location of disturbances such as deforestation events across the earth surface. Typical scenarios require the analysis of billions of image patches/pixels. While various change detection techniques have been proposed in the literature, the associated implementations usually do not scale well, which renders the corresponding analyses computationally very expensive or even impossible. In this work, we propose a novel massively-parallel implementation for a state-of-the-art change detection method and demonstrate its potential in the context of monitoring deforestation. The novel implementation can handle large scenarios in a few hours or days using cheap commodity hardware, compared to weeks or even years using the existing publicly available code, and enables researchers, for the first time, to conduct global-scale analyses covering large parts of our Earth using little computational resources. From a technical perspective, we provide a high-level parallel algorithm specification along with several performance-critical optimizations dedicated to efficiently map the specified parallelism to modern parallel devices. While a particular change detection method is addressed in this work, the algorithmic building blocks provided are also of immediate relevance to a wide variety of related approaches in remote sensing and other fields.**

I. Introduction

In the context of climate change, estimating and reducing global deforestation is crucial to mitigate the effect of global warming [1]. This makes mapping forest disturbances at large scales more important than ever before [2]. The availability of high-quality and detailed satellite data provides a huge opportunity to detect and monitor deforestation across the pantropics, which usually happen over large and often inaccessible areas, see Figure 1. In this context, the timely and spatially accurate detection of such events is critical to enable a better protection and to trigger countermeasures (e.g., in the context of deforestation currently occurring in the Brazilian Amazon).

Time-series based change detection of satellite data has matured to a well-established tool for monitoring such changes and disturbances in terrestrial ecosystems [3]. Such change detection methods typically operate on individual pixels to detect "changes" and have been successfully applied for a variety of different applications [4]–[15]. One of the state-of-the-art methods is the so-called *break detection for additive season and trend* (BFAST) approach, which combines time series based break detection with seasonal-trend modelling [16]. This approach has been used across various domains and for a wide variety of time series based change monitoring tasks (e.g., urban expansion [17], water extent changes [18], [19], burned area detection [20], or biomass estimation [21]).

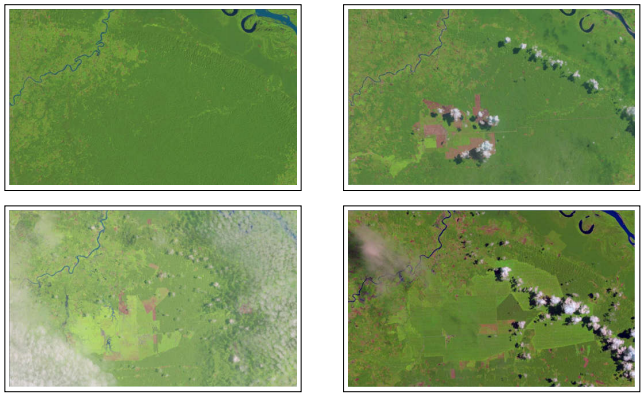


Fig. 1: Deforestation visible in a sequence of satellite images obtained in 1996, 2007, 2009, and 2018. Typically, sequences with hundreds of images are considered, which yield, for each individual pixel, a time series of reflectance measurements.

While such change detection schemes yield robust results, the induced fitting processes can become a major computational bottleneck even if only few satellite images are considered [4], [5]. This has limited the application of such change detection methods to relatively small areas and scenarios. In recent years, the quality and availability of remote sensing data have changed dramatically, starting with the Landsat mission [22] that made imaging data with a spatial resolution of 30 m freely available at a global scale. Such projects yield petabytes of data every year with hundreds of billions of pixels to be analyzed [22], [23]. For data volumes at this scale, the application of the aforementioned change detection schemes has—so far—required immense computational resources, which, in turn, has rendered the analyses at continental/global scales infeasible in practice.

This work addresses this computational bottleneck by providing an efficient massively-parallel implementation for BFAST (more precisely, for BFAST-Monitor), which can effectively handle scenarios with billions of pixels. Our approach

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00040

is three orders of magnitudes faster than the commonly used implementation and allows researchers, for the first time, to handle massive change detection scenarios covering large parts of our Earth using little computational resources. Our implementation is made publicly available and will be useful for numerous applications in remote sensing [4]–[15]. The particular contributions of this work are described in detail in Section II-B after presenting the relation with related work.

II. Background and Contributions

A. Unsupervised Change Detection for Satellite Data

Modern satellites collect multi-spectral image data over time. For instance, in the case of the Sentinel 2 mission [23], up to 13 (grayscale) images are obtained for each scene every five days (Figure 1 shows RBG composites that are based on such images). Such time series of satellite data often form the basis for "change" or "break" detection tasks. Typically, the image data are preprocessed and so-called *vegetation indices* are derived. More specifically, for detecting changes in forest cover, wetness-related spectral vegetation indices such as the *Normalized Difference Moisture Index (NDMI)* are used to extract quantitative information on the amount of vegetation for every pixel [24]. Given such preprocessed images, semi-automatic change detection algorithms are typically applied per pixel on a time series consisting of NDMI values per pixel.

In the literature, a variety of different yet related methods can be found. A common feature of those methods is the fact that they often fit "simple" regression models per pixel time series and that discrepancies are analyzed between the models and the true data [4], [9]–[14]. The BFAST-Monitor approach [16] is among the most well-known methods in this context, see Figure 2 for an illustration of its application on time series data for two different pixels. This approach fits a linear regression model based on a period defined as *stable history period*. Next, statistical break detection is performed on the data in a so-called *monitoring period* [25], i.e., the period that shall be analyzed for changes (e.g., if a deforestation event has occurred or not). If the difference between the model and the data in the monitoring period is significant, a *break* is detected. The *change magnitude* is recorded as the mean deviation between the model and the data [16]. More formally, given a time series $y_1, \ldots, y_N$ for a pixel, a model of the form

$$\hat{y}_t = \alpha_1 + \alpha_2 t + \sum_{j=1}^{k} \gamma_j \sin\left(\frac{2\pi j t}{f} + \delta_j\right) + \epsilon_t \qquad (1)$$

is assumed. Here, the first and second term determine the intercept and the trend, respectively. The third term specifies the seasons. More specifically, the time series data are modeled via *amplitudes* $\gamma_1, ..., \gamma_k$, and the *phases* $\delta_1, ..., \delta_k$ (i.e., seasons). The parameter f specifies the *frequency* of the observations. For instance, $f = 365$ for time series with an interval of one day between the different observations or $f = 23$ for time series with an interval of 16 days. The parameter k determines the number of *harmonic terms* that capture the seasonal pattern (typically very small, e.g., $k = 2$ or $k = 3$). The remaining error is captured by ϵ_t at time $t = 1, \ldots, N$.

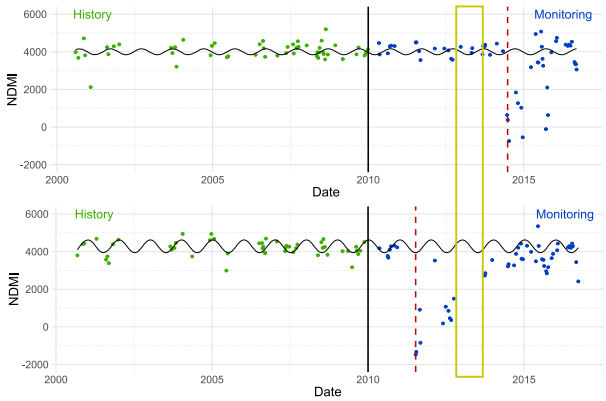


Fig. 2: Application of a break detection algorithm [16] on time-series data from two locations (pixels). For each time series, a model is derived based on data given for a history period (green points). In case the predictions generated by this model are not in line with the data available for the monitoring phase (blue points), a "break" is detected (red dashed line). Note that the time series usually vary both w.r.t. the number and location of measurements (yellow rectangle). For large-scale experiments, millions or even billions of such models have to be computed. Currently available implementations cannot handle such scenarios or require significant computational resources. Our work addresses this computational bottleneck and provides an efficient parallel (GPU) implementation that can deal with large datasets given limited resources.

The model given by Equation (1) can be written as a standard linear model of the form

$$\hat{y}_t = \mathbf{x}_t^\top \boldsymbol{\beta} + \epsilon_t \qquad (2)$$

with patterns $\mathbf{x}_t$ being defined as

$$\mathbf{x}_t = (1, t, \sin(F_t(1)), \cos(F_t(1)), \ldots, \sin(F_t(k)), \cos(F_t(k)))^\top \qquad (3)$$

that are generated for each date t, where $F_t(j) = {}^{2\pi j t}/_{f}$.

The stable history period consists of the first n elements, and the monitoring period is based on the remaining measurements $y_{n+1}, \ldots, y_N$. The second part is used to "test" for changes. To measure the discrepancy between the model and the measurements in the monitoring period, one resorts to a *moving sums* (MOSUM) process

$$MO_t = \frac{1}{\hat{\sigma}\sqrt{n}} \sum_{s=t-h+1}^{t} \left(y_s - \mathbf{x}_s^\top \hat{\boldsymbol{\beta}}\right), \qquad (4)$$

with a user-defined *bandwidth* $1 \leq h \leq n$ and $\hat{\sigma} = \sqrt{\sum_{i=1}^{n} r_i^2 / (n-2)\cdot(k+1)}$. If the observations of the monitoring period are similar to those of the history period, the process should stay within the bounds b_t specified by the user and no break should be detected (e.g., $b_t = \lambda\sqrt{\log_+ {}^{t}/_{n}}$ for some $\lambda > 0$). Otherwise, a break is detected, see again Figure 2.

Thus, the break detection is conducted per pixel by defining a dataset $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\} \subset \mathbb{R}^K \times \mathbb{R}$ with $K = 2 + 2k$, which contains, for each date t, the constructed

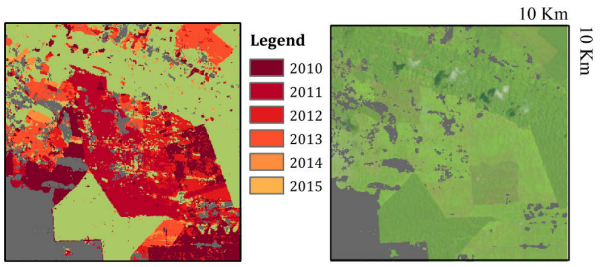


Fig. 3: The left figure illustrates changes detected by applying BFAST-Monitor for an area in Peru, where clear-cut deforestation due to palm oil plantations occurred. The monitoring period considered was 2010–2015, masking out from the analysis areas that were not forest at the moment of 2010 (gray coloured in figures). While the deforestation is visible also in the RGB satellite image (right), detecting these changes using BFAST-Monitor (left) offers an insight on when these events occurred, and makes them quantifiable in time and space.

pattern $\mathbf{x}_t$ defined in Equation (3) along with the pixel value y_t. Afterwards, a linear model is computed via

$$\underset{\boldsymbol{\beta}\in\mathbb{R}^K}{\text{minimize}} \sum_{i=1}^{n} \left(y_i - \mathbf{x}_i^\top \boldsymbol{\beta}\right)^2, \tag{5}$$

which can be used to obtain the MOSUM process (4) and the break points. To analyze if changes have occurred in a certain area, such a model and the associated MOSUM process are computed for each pixel, see Figure 3.

It is worth stressing the following. First, the only input required for the application of BFAST-Monitor is the start of the monitoring period (i.e., the index $n+1$), the number k of harmonic terms, as well as some other parameters (e.g., the size h of the moving window), for which one can often resort to default values. However, no labels for breaks or changes are required, i.e., the approach is unsupervised. This is different from supervised change detection algorithms based on, e.g., deep learning, which typically require large amounts of labeled data (see, e.g., Reichstein *et al.* [26]). BFAST-Monitor and related schemes can therefore be seen as some kind of unsupervised anomaly detection schemes. Second, the pixel-wise fitting of "simple" regression models is the main building block for a variety of such unsupervised change detection approaches [4], [9]–[14]. For this reason, the implementation and the algorithmic building blocks provided in this work are of immediate relevance for those schemes as well.

B. Contributions

While recent work has demonstrated the potential of high-performance computing for accelerating such break detection schemes [27], a simpler scenario was considered, which is based on the assumption that all time series values are valid, i.e., that all pixels yield time series having the *same* length with patterns $\mathbf{x}_t$ generated for the *same* dates t. This regularity could be exploited for the computation of the individual models (e.g., the inverted squared data matrix—see Algorithm 1—is the same for all pixels in this special case and has to be computed only once for *all* the pixels). Unfortunately, this restriction makes it impractical for many vegetated areas on our Earth, where clouds frequently mask image pixels. For instance, in Figure 1, many pixels would be masked out, since they are occluded by clouds in at least one image. Such pixel values would be ignored, which, in turn, would lead to the individual time series having a different length with "missing values" occurring at different times, see again Figure 2.

Our approach is fundamentally different and can handle the general and much more common case. In particular, we provide a massively-parallel implementation that can be used for change detection scenarios with potentially many missing values per time series. From a computational perspective this case is considerably harder to parallelize, since the single tasks are very *irregular* in the sense that one is given very different time series for the individual pixels with both different computational demands and different memory access patterns. Overall, we make the following contributions:

- We present a high-level specification of our parallel algorithm that describes all available (nested) parallelism. We also report a fully-automated compilation strategy that effectively maps the specified parallelism to modern graphics processing units (GPUs).
- We propose several performance-critical optimizations that result in speed-ups as high as $6\times$ at individual-kernel level and as high as $4.5\times$ at application level. The GPU implementation is $24-48\times$ faster than a corresponding multi-threaded `C` implementation using 32 threads. We also provide detailed experimental results that compare the performances both across different datasets and between the different optimization recipes.
- We also provide results for a large-scale experiment covering the entire continental tropical Africa—a scenario that was basically impossible to handle before.

Our implementation is up to three orders of magnitudes faster than the commonly used `R` implementation.[1] It allows researchers, for the first time, to apply the change detection schemes outlined above for data covering large parts of our Earth using little computational resources (e.g., a single desktop computer). Finally, we would like to mention that all optimizations were implemented as general code transformations in the Futhark language [28]. In fact all code versions were automatically compiled from high-level Futhark specifications. This will significantly facilitate the adaptation of our framework to other related change detection methods.

C. Related Work

A variety of anomaly detection methods can be found in the data mining literature, see, e.g., Aggarwal [29], [30] for an overview. The focus of our work is, however, on accelerating the computations needed by the aforementioned *unsupervised*

[1] Available under https://bfast2.github.io. A Python package containing our implementation is available under https://github.com/gieseke/bfast.

change detection methods—in particular BFAST-Monitor—which are commonly used in remote sensing [4]–[8], [15], [17]–[21]. As already pointed, these methods do not require any labels for the individual pixels and typically resort to a history and monitoring period for fitting the models. It is worth stressing that the focus of this work is *not* on supervised change detection schemes (or, more general, landcover classification methods), which require labels for each pixel/patch that is considered and which are nowadays typically addressed via deep learning [26].

To the best of our knowledge, all existing unsupervised change detection implementations resort to high-level programming languages such as Python or R with the computations potentially parallelized over the individual pixels [31], and "most of the applications [of such change detection implementations] are limited to small areas due to constraints of storage and computing resources" [3]. To address large scenarios, the typical approach in the remote sensing field is to resort to significant compute resources. For instance, Bullock *et al.* [14] combine three such approaches to improve the accuracy of the change detection and report that the analysis of three Landsat scenes (each with 6300×6000 pixels) took 48 hours on a "200 high-speed node machine" [14]. This work is the first that provides a parallel implementation for unsupervised change detection frameworks given irregular time series data with potentially many missing values, which runs efficiently on commodity GPU hardware.

III. Algorithmic Framework

A. Detecting Breaks in Time Series with Missing Values

The algorithmic building blocks of BFAST-Monitor are provided in Algorithm 1: For each individual time series, one is given a vector $\mathbf{y}$ containing the target values for both the history and the monitoring period. The corresponding data matrix $\mathbf{X}$ consists of the patterns $\mathbf{x}_i$ defined in Equation (3). As mentioned above, many of the $\mathbf{y}_i$ values might be missing. Those values are filtered out by the function FilterMissing in Line 1, yielding a new target vector $\overline{\mathbf{y}}$ and a new data matrix $\overline{\mathbf{X}}$ for each time series not containing any missing values anymore. The original indices are stored in the array $\overline{\mathbf{I}}$, which is used to remap the indices at the end. Both $\overline{\mathbf{y}}$ and $\overline{\mathbf{X}}$ are used to fit a standard linear regression model

$$\overline{\boldsymbol{\beta}} = \overline{\mathbf{M}}^{-1}\overline{\mathbf{X}}_{[:,:\overline{n}]}\overline{\mathbf{y}}_{[:\overline{n}]} \tag{6}$$

for the history period, where $\overline{n}$ specifies the new end of the history period in $\overline{\mathbf{X}}$.[2]

The residuals $\overline{\mathbf{r}}$, computed in Line 5, are used to obtain the MOSUM process $\overline{\mathbf{m}}$. These values along with the boundary values $\overline{\mathbf{B}}$ are then used to compute the desired break indices in Line 12. The final step takes care of remapping these indices such that the indices are in line with the target input values

[2]We use the following notation: $\mathbf{M}_{[:n,:m]}$ denotes the submatrix of $\mathbf{M}$ only containing the first n rows and m columns, $\mathbf{M}_{[:,:m]}$ the submatrix containing the first m columns, and $\mathbf{M}_{[:n,:]}$ the submatrix containing the first n rows. Similarly, $\mathbf{a}_{[:n]}$ denotes the vector containing the first n elements of a vector $\mathbf{a}$.

Algorithm 1 BFAST-Monitor

Require: A vector $\mathbf{y} = (y_1, \ldots, y_N)^\top \in \mathbb{R}^N$ containing a time series with $0 \leq s < N$ missing values and the complete data matrix $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N) \in \mathbb{R}^{K\times N}$.

Ensure: A vector $\mathbf{b} = (b_{n+1}, \ldots, b_N)^\top \in \{0,1\}^{N-n}$ containing the detected breaks for the monitoring period.

1: $\overline{\mathbf{y}}, \overline{\mathbf{X}}, \overline{\mathbf{I}}, \overline{n}, \overline{N}$ = FilterMissing($\mathbf{y}, \mathbf{X}, n, N$)
2: $\overline{\mathbf{M}}$ = Multiply($\overline{\mathbf{X}}_{[:,:\overline{n}]}, (\overline{\mathbf{X}}_{[:,:\overline{n}]})^\top$) ▷ $\overline{\mathbf{M}} \in \mathbb{R}^{K\times K}$
3: $\overline{\mathbf{M}}^{-1}$ = InvertMatrix($\overline{\mathbf{M}}$) ▷ $\overline{\mathbf{M}}^{-1} \in \mathbb{R}^{K\times K}$
4: $\overline{\boldsymbol{\beta}} = \overline{\mathbf{M}}^{-1}\overline{\mathbf{X}}_{[:,:\overline{n}]}\overline{\mathbf{y}}_{[:\overline{n}]}$ ▷ $\overline{\boldsymbol{\beta}} \in \mathbb{R}^K$
5: $\overline{\mathbf{r}} = \overline{\mathbf{X}}^\top\overline{\boldsymbol{\beta}} - \overline{\mathbf{y}}$ ▷ $\overline{\mathbf{r}} \in \mathbb{R}^N$
6: **for** $t = \overline{n}+1, \ldots, \overline{N}$ **do**
7: $\quad\overline{\mathbf{m}}[t-\overline{n}-1] = \frac{1}{\hat{\sigma}\sqrt{\overline{n}}} \cdot \sum_{i=t-h}^{t} r_i$ ▷ $\overline{\mathbf{m}} \in \mathbb{R}^{\overline{N}-\overline{n}}$
8: **end for**
9: **for** $t = \overline{n}+1, \ldots, \overline{N}$ **do**
10: $\quad\overline{\mathbf{B}}[t-\overline{n}-1] = \lambda\sqrt{\log_+ t/\overline{n}}$ ▷ $\overline{\mathbf{B}} \in \mathbb{R}^{\overline{N}-\overline{n}}$
11: **end for**
12: $\overline{\mathbf{b}} = |\overline{\mathbf{m}}| > \overline{\mathbf{B}}$ ▷ $\overline{\mathbf{b}} \in \mathbb{R}^{\overline{N}-\overline{n}}$
13: $\mathbf{b}$ = RemapIndices($\overline{\mathbf{b}}, \overline{\mathbf{I}}$) ▷ $\mathbf{b} \in \mathbb{R}^{N-n}$

$\mathbf{y}$ (which can contain missing values). In addition to these indices, the mean of the MOSUM process is computed and returned as well (not shown in the algorithm).

The model $\boldsymbol{\beta}$ for a single pixel can be obtained in $\mathcal{O}(K^3 + K^2 n)$ time, where $\mathcal{O}(KN)$ time is needed for computing the predictions for the monitoring period and the steps to identify the breaks. While being computationally not very demanding for a single pixel, the involved computations become extremely challenging in case scenarios with billions of pixels are addressed. As explained before, not all measurements y_i are usually available for a certain pixel, which generally leads to many different individual regression and monitoring tasks. In particular, assuming an arbitrary amount of missing values per time series, one is faced with up to $\sum_{k=1}^{n} \frac{n!}{k!}$ possible realizations for the data matrices $\overline{\mathbf{X}}$, which cannot all be stored or precomputed in practice. These matrices all form sub-matrices of $\mathbf{X}$ and can, for each pixel, be extracted via FilterMissing in Line 1 of Algorithm 1. However, the following steps (e.g., Steps 2–4) depend on the particular $\overline{\mathbf{X}}$ and have, hence, to be individually computed for each pixel.

B. High-Level Parallel Specification

Algorithm 1 shows the main computational steps for an individual pixel time series, but in practice, we aim to process a large batch of image pixels simultaneously. This is simply achieved by wrapping the computation in *an outer parallel loop*—because the computation of any pixel is independent of any other, e.g., each pixel reads the shared data matrix $\mathbf{X}$ and its corresponding target vector $\mathbf{y}$, while the intermediate and result arrays are private to each pixel.

The difficulty consists in deciding *how to best exploit the inner parallelism* available in the computations specific to each pixel. For example, matrix-matrix and matrix-vector

multiplication (Lines $2, 4, 5$), matrix inversion (Line 3), vector subtraction (Line 5) are well-known parallel operations. Similarly, the loop between Lines $9-11$ and the vector operations in Lines 12 and 13 are also trivially parallelizable. Finally, the filtering operation in Line 1 and the loop between Lines $6-8$ can also be parallelized, albeit they require non-trivial rewriting that uses parallel-prefix sum operators [32]. A further complication is that the inner-parallel operations have *different sizes*, but the GPU hardware provides morally a flat-parallel programming interface. This renders significant rewriting necessary to map the available parallelism to a form that GPUs can exploit, as detailed next.

1) Extreme Parallelization Strategies: One extreme of the design space is to (efficiently) sequentialize the inner parallelism, by mapping the per-pixel computation to one (CUDA) thread.[3] Besides the ease of implementation, this approach has the advantage that it can more aggressively fuse the parallel operations into sequential loops in a manner that significantly reduces the number of accesses to global memory (which are up to two-orders of magnitude more expensive than scalar operations). The downsides are that (i) temporal locality cannot be optimized beyond what the magic of hardware caches can offer, (ii) thread divergence[4] limits the gains of operating on the logical size of the filtered arrays—because $\overline{n}$ and $\overline{N}$ differ across pixels, and (iii) hardware might be underutilized if the parallel batch is not large enough.

The other extreme is to apply a well-known transformation that flattens all parallelism [32], such that it can be mapped to hardware. This preserves asymptotically the number of operations of the nested-parallel program, but at the expense of (i) compromising temporal locality, (ii) accessing global memory more often, and (iii) introducing many prefix-sum operations, which are less efficient on GPU than parallel loops. Furthermore, there are no easy ways or standard tricks to optimize the resulted flattened code (other than loop fusion).[5]

2) Our Parallelization Strategy: Our strategy is a midpoint between the two strategies sketched above: we distribute the outer parallel loop (of pixels) around (groups of) operations of same (inner-parallel) size, and pad them to the maximal size across all threads/pixels,[6] such that we can translate each such group to a (CUDA/OpenCL) kernel. For example, Line 2 of Algorithm 1 will be mapped to a batched matrix-matrix multiplication-like kernel, Line 3 to a batched-matrix inversion kernel, and Lines 4 and 5 to three matrix multiplication-like kernels, etc. The downside of this approach is the overhead introduced by padding. The advantage is that it enables a systematic optimization of temporal locality for each kernel. For example, non-trivial tiling techniques can be applied for (batched) matrix multiplication. For the other kernels, the CUDA block size is chosen equal to the inner-parallel size—i.e., a pixel is processed by a CUDA block—which allows the intermediate arrays to be explicitly stored and reused from fast/scratchpad memory (shared memory in CUDA).

For completeness, we provide the full data-parallel hardware independent specification written in the Futhark language [34] in the appendix (Figure 12). For readability purposes, we have organized the discussion of parallelization strategies around well-known matrix operations, such as matrix multiplication and inversion. It is important to remark, however, that about half of the execution time is spent in kernels (`ker 7,8,9,10` in Figure 12) that do not correspond to such matrix operations. The proposed optimizations highlighted in the next section are not restricted to such matrix operations; for example, kernels `ker 7,8,9,10` are optimized in a similar way as matrix inversion.

C. Performance-Critical Optimizations

This section discusses two illustrative and performance-critical optimizations. Their impact at kernel and application level is demonstrated in Section IV. The optimized computational kernels are (1) a special kind of batch matrix multiplication, in which the same two matrices are multiplied, but in which the values of the two matrices are filtered under a mask that differs across the batch, and (2) a batch matrix inversion for "small" matrices (e.g., 16×16).

1) Register Tiling of Batch Matrix Multiplication (Like): Figure 4a shows `C`-like pseudocode corresponding to computing Line 2 of Algorithm 1 for `M` pixels (time-series), where **`forall`** and **`forseq`** denote parallel and sequential loops, respectively. Matrices `A` and `B` correspond to $\mathbf{X}_{[:,:n]}$ and $\mathbf{X}^T_{[:,:n]}$ of size `K×n` and `n×K`, respectively—but the transformation is valid for different `K`s, i.e., $\texttt{K}_1$ and $\texttt{K}_2$ in the figure. The important thing to notice is that the matrix $\mathbf{X}$ has *not* been filtered—its inner size is `n` rather than $\overline{n}$. Instead, the innermost sequential loop has been padded to count `n` and the missing values are not accumulated—i.e., `1-isnan(Y[i,q])` evaluates to `0` under the convention that a missing value is represented by `NaN`. *It is worth noting that such a code is only akin to matrix multiplication—due to the filtering under the* `Y` *mask—but, to our knowledge, it is not yet supported by any GPU or CPU high-performance library.*

The code exhibits a key pattern that allows to significantly optimize temporal locality: the subscripts of any of the matrices `A`, `B`, and `Y` are invariant to exactly two of the three outer-parallel loops—e.g., $\texttt{A[j}_1\texttt{,q]}$ is invariant to the loops of indices `i` and $\texttt{j}_2$, and `Y[i,q]` to the loops $\texttt{j}_1$ and $\texttt{j}_2$. The result of the transformation is shown in Figure 4b. The essence of it is that the original three (parallel) loops have

[3]We use CUDA terminology [33] for all GPU programming aspects.

[4]On NVIDIA GPUs a (half-) warp of threads execute in lockstep; it follows that if the threads in the same warp execute loops of different counts, then they will all have to wait for the thread executing the largest loop before advancing to the next computation.

[5]For example, denoting by M the number of pixels, flattening the padded version of matrix multiplication at Line 2 in Algorithm 1 would require $3{\cdot}M{\cdot}n{\cdot}K^2$ accesses to global memory (with fusion). If filtering is performed and assuming that 90% of values are missing, it would require at least $4.5{\cdot}M{\cdot}n{\cdot}K^2$—because filtering the missing values requires two scan operations applied on arrays of size $m{\cdot}n{\cdot}K^2$, which already cost $4{\cdot}M{\cdot}n{\cdot}K^2$ accesses—and $0.4{\cdot}M{\cdot}n{\cdot}K^2$ extra memory. In comparison, the technique used in this work—described in Section III-C—requires no extra memory and has only $\frac{3{\cdot}m{\cdot}n{\cdot}K}{30}$ accesses to global memory, i.e., about $30\times$ and $45\times$ fewer global-memory accesses than the flattened versions, respectively.

[6]For example, the (inner) loop between Lines $9-11$ of Algorithm 1 has logical count $\overline{N}-\overline{n}$, which we pad to its upper bound $N-n$.

```
forall(i=0; i<M; i++) {          // parallel
  forall(j1=0; j1<K1; j1++) {    // parallel
    forall(j2=0; j2<K2; j2++) { // parallel
      float acc = 0.0;
      forseq(q=0; q<n; q++) {
        float a = A[j1,q], b = B[q,j2], ab=a*b;
        acc += ab * (1.0 - isnan(Y[i,q]));
      }
      M̄[i,j1,j2] = acc;
} }
```

(a) Naive Imperative Version

```
Y^T = transpose(Y);
forall(ii=0; ii<M; ii+=R) {          // grid.z
  forall(jj1=0; jj1<K1; jj1+=T1) {   // grid.y
    forall(jj2=0; jj2<K2; jj2+=T2) { // grid.x
      forall(j1=jj1; j1<min(jj1+T1,K1);
             j1++) {                 // block.y
        forall(j2=jj2; j2<min(jj2+T2,K2);
               j2++) {               // block.x
          float Y^q_sh[R];        // shared memory
          float acc[R];             // registers
          for(i=0; i<R; i++)    // fully unroll
            acc[i] = 0.0;
          float a, b, ab, y;
          forseq(q=0; q<n; q++) { barrier;
            float a=A[j1,q], b=B[q,j2], ab=a*b;
            // collective copy global-to-shared
            Y^q_sh[0:R] = Y^T[q,ii:min(ii+R,M)];
            barrier;       // block-level synch
            forseq(i=0; i<R; i++){ // fully unroll
             if(ii+i < M)
              acc[i] += ab*(1.0-isnan(Y^q_sh[i]));
          } }
          forseq(i=0; i<R; i++)// fully unroll
           if (ii+i<M) M̄[ii+i,j1,j2] = acc[i];
} } } } }
```

(b) Register Tiled Implementation

Fig. 4: Batched Matrix Multiplication under Variant Y Mask.

been tiled[7] with tile sizes R=30 and $T_{1/2}$=min(16,$K_{1/2}$). The resulting three parallel outer loops of indices ii, jj_1, jj_2 morally form the CUDA grid, while their $T_{1/2}$ tiles—i.e., the loops of indices j_1 and j_2—form the CUDA block.[8]

The R-tile loop of index i has been sequentialized and moved in the innermost position in the nest by distributing it across the statements of the original loop-nest body. The distribution requires to expand the local variables that are updated with values variant to loop i with an extra array dimension. For example acc was previously a scalar, but after distribution it has become a length-R array of floats, but which is actually stored in registers. In contrast, a and b have been hoisted outside loop i (and not expanded) because their values (e.g., A[jj_1+j_1,q]) are invariant to loop i.

The transformed code promotes temporal locality because it performs a factor of R× fewer accesses to arrays A, B and Y, which are necessarily stored in global memory:

1. the computation of ab has been hoisted outside the loop of index i, and hence one global-memory access to A/B is amortized by R accesses to the register holding ab;
2. a collective copy performed in parallel by the threads in the CUDA block brings the slice Y^T[q,ii:ii+R] from global to fast (CUDA-shared) memory, from where it is also reused R times (transposing Y in Y^T was necessary to ensure coalesced accesses in the copying operation).

[7]Stripming a loop for(i=0; i<M; i++) body with a tile R corresponds to re-writting the original loop as two nested loops in which the outer one advances with stride R and the inner one with stride 1, i.e., for(ii=0;ii<M;ii+=R){for(i=ii;i<min(M,ii+R);i++)body}. Tiling corresponds to stripmining a loop, followed by interchanging its tile in an inner position in the original nest.

[8]Threads within a CUDA block can be synchronized by barriers and can utilize scratchpad/fast (shared) memory as a software-managed cache.

```
float[M][K][K] batchMatInv(float A[M][K][K]) {
  float A^-1[M][K][K];   // global memory result
  forall i = 0...M-1 {                 // grid.x
    float A_sh[K][2*K];          // shared memory
    // Pad A with identity matrix to the right
    forall k1 = 0...K-1 {              // block.y
      forall k2 = 0...2*K-1 {          // block.x
        if (k2<K){ A_sh[k1,k2] = A[i,k1,k2];  }
        else     { A_sh[k1,k2] = (k2 == K+k1);}
        barrier;           // block-level synch
    } } // end forall k1/2
    // Gauss-Jordan Elimination:
    forseq q = 0...K-1
      float vq = A_sh[0,q]
      forall k1 = 0...K-1 {            // block.y
        forall k2 = 0...2*K-1 {        // block.x
          float tmp = 0.0;
          if (vq == 0.0) tmp = A_sh[k1,k2];
          else {
            float x = A_sh[0,k2] / vq;
            if(k1 == K-1) tmp = x;
            else tmp = A_sh[k1+1,k2] -
                       A_sh[k1+1,q] * x;
          }
          barrier;         // block-level synch
          A_sh[k1,k2] = tmp;
          barrier;         // block-level synch
      } } // end forall k1/2
    } // end forseq q
    // collective copy shared-to-global mem:
    A^-1[i, 0:K, 0:K] = A_sh[0:K, K:2*K];
} } // end forall i
```

Fig. 5: Batched Matrix Inversion: Loop-Based Specification

2) Batched Matrix Inversion in Shared Memory: The next optimization addresses batched matrix inversion. Figure 5 shows C-like pseudocode for computing the inverses of a size-M batch of K×K matrices, stored in the global-memory array A, by means of Gauss-Jordan elimination. The batch dimension—the loop of index i—is mapped to the CUDA grid, and the i^{th} CUDA block is responsible to invert matrix A[i], where the block consists of K×(2K) threads, and are represented by the parallel loop nests of indices $k_{1,2}$.

The computation proceeds by semantically adjoining the identity matrix to the right side of A[i], and by storing the result in array A_{sh}, which is allocated in fast (shared) memory. Then each iteration of the sequential loop of index q updates in parallel all elements of the matrix A_{sh}, until the left side of A_{sh} is reduced to the identity matrix—this is guaranteed to happen

in maximum K sequential steps. Finally, the inverted matrix is the right-hand side of the adjoined matrix (i.e., A_{sh}[0:K, K:2*K]), which is collectively copied by the threads of the block to the result matrix A^{-1}[i], stored in global memory.

This example demonstrates the benefits of our strategy, described in Section III-B, which distributes the computation for a batch of pixels across each group of same-size (inner) parallel operations. This allows to adjust the CUDA-block size for each kernel to match the inner-parallel size of that kernel; in this case K×2·K. At its turn, this allows (i) to allocate array A_{sh} in shared memory—which offers significantly reduced latency than global memory—and (ii) to repeatedly use and update its values within a sequential loop of count K. Our optimized implementation performs a factor of $3K$ fewer accesses to global memory in comparison to a "naive" implementation, which generates a factor of $5 - 6\times$ speed-up.

D. Implementation Details

The overall implementation of Algorithm 1 is based on Python (version 3.6) and resorts to the optimized kernels outlined above, which are integrated via PyOpenCL [35]. All GPU code is generated automatically from data-parallel hardware-agnostic Futhark specifications, similar to the one provided in the appendix (Figure 12). The data are usually provided as GeoTIFF files, which contain the compressed images. In case the uncompressed image data are too large to fit in host/GPU main memory, they first get split into chunks (which happens on the host). For each chunk, the data are copied from host to GPU prior to conducting some preprocessing steps (e.g., data-dependent initialization of parameters or removal of slices only containing NaN values). Afterwards, the GPU kernels are invoked via PyOpenCL, followed by copying the results (e.g., the break indices **b**) back to the host.

IV. Performance Analysis

This section evaluates on synthetic datasets the impact of (i) the proposed transformations at individual-kernel level, and (ii) the compilation strategy at application level, respectively.

A. Experimental Setup and Datasets

All experiments described in this section were performed on an Intel system with 128GB RAM, 16 Xeon cores, model E5-2650 v2, running at 2.60GHz, using 2-way hyperthreading, which is also equipped with an RTX2080Ti NVIDIA GPU with 11GB DRAM, 4352 cores running at 1.55GHz under CUDA 9.2. The CPU-parallel code is hand-written in C and compiled with GCC 4.8.5 (-fopenmp -O2); the reported CPU runtimes were averaged across 20 runs and are based on 32 threads. We measured the total application runtime, minus the time taken for initializing the GPU context, for loading the program input onto the GPU, and for copying the results back to the host; excluding these fixed overheads emphasizes the performance differences between the kernel implementations. The reported GPU runtimes were averaged across 250 runs.

We report performance both in milliseconds—the best GPU runtime is displayed under the x-axis in the performance figures—and in *specification-giga floating-point operations (flops) per second*, dubbed GFlops^{Sp}. The latter computes the worst-case number of flops directly from the high-level specification by means of an algebraic formula, which is written in terms of the dataset-specific sizes M, N, n, K, and which assumes that all flops have unit cost, including special functions such as sqrt. In essence, GFlops^{Sp} measures a notion of normalized runtime, which allows to meaningfully compare performance both between differently-optimized code versions and across different datasets.

	D1	D2	D3	D4	D5	D6	Peru (Small)	Africa (Small)
M	16384	16384	32768	32768	65536	16384	111556	589824
N	1024	512	512	256	256	1024	235	327
n	512	256	256	128	128	256	113	160
f^{NaN}	50%	50%	50%	50%	50%	75%	69%	92%

TABLE I: The parameters M, N, n, and f^{NaN} denote the number of pixels, the lengths of the time series, the length of the history period, and the frequency of (NaN) values, respectively.

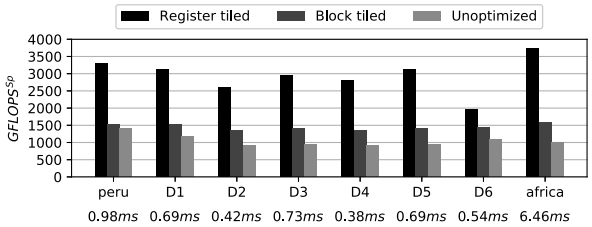


Fig. 6: Performance of Batch-Masked Matrix Multiplication

The performance was analyzed on the datasets shown in Table I; their parameters M, N, n are named as in Algorithm 1, where M denotes the number of image pixels and where f^{NaN} denotes the frequency of invalid (NaN) values. Datasets D1–D5 are thought to vary the values for M and N, while keeping n = N/2. The rationale of these datasets is to demonstrate that the performance is relatively stable in such a context. Dataset D6 is intended as an adversarial/stress case: It has a rather small n = N/4 and a quite high NaN frequency. While datasets D1–D6 are artificial[9], Peru(Small) and Africa(Small) are real-world datasets, which exhibit a high frequency of invalid values, f^{NaN}=69% and 92%, respectively. In spite of the high f^{NaN}, the performance on Peru(Small) and Africa(Small) is competitive with D1–D5 and superior to D6, which indicates that the synthetic datasets are representative (if slightly pessimistic) test cases. For all experiments, we resort to k=3 and thus K=2·k+2=8, which is a common case in practice (also, due to tiling, larger k values result in higher performance).

B. Impact of Optimizations on Individual Kernels

Figure 6 compares the performance of three implementations for the batch computation that squares matrix $\mathbf{X}_{[:,:n]}$ under the mask given by the time series of each pixel $Y_{[:,:n]}$, as

[9]We use artificial datasets since we can control their characteristics and can keep their sizes small enough to allow the experimental suite to run in a reasonable runtime.

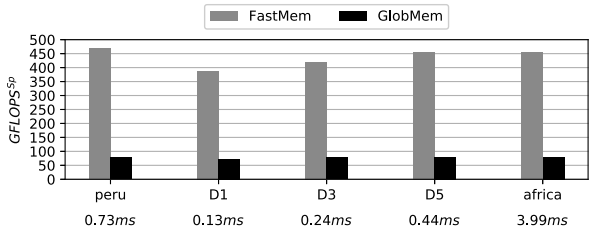


Fig. 7: Performance of Batched Gauss-Jordan Matrix Inversion

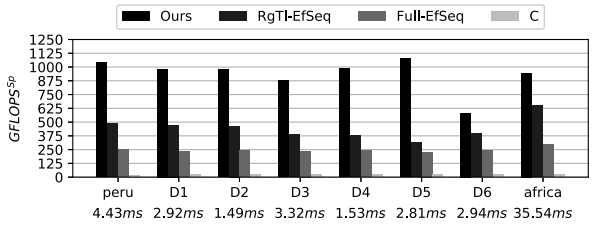


Fig. 8: Performance of one GPU invocation

presented in section III-C1. The number of flops is $4MnK^2$. The first bar in the figure refers to our register-tiling contribution, the second bar refers to two-dimensional block tiling (supported by the Futhark compiler [34]), and the third bar refers to no tiling at all, i.e., unoptimized).

The performance of register tiling is relatively stable between $2.6 - 3.7$ TFlops^{Sp} on all datasets except for `D6`. The poorer performance on `D6` is due to a compiler inefficiency that results in the whole matrix `Y` being transposed rather than only the slice corresponding to the training set $\texttt{Y}_{[:M,:n]}$. This affects all datasets, but predominantly `D6`, which has `n=N/4`, while the others have `n=N/2`. We remark that solving that inefficiency would not only raise the performance of `D6` to a level similar to the others, but will generate an additional $\sim 1.25\times$ speed-up for the other datasets. In terms of the whole application runtime, the discussed (transposition) inefficiency is responsible for a $\sim 5\%$ slowdown (and $\sim 8\%$ for `D6`).

In comparison with the other two code versions—which do not require transposition for `Y`—the register tiling currently outperforms them by a $2 - 3\times$ factor. One can observe that, for this kernel, block tiling offers limited performance gains over unoptimized code, because the temporal locality of `Y` is not optimized (i.e., accessed repeatedly from global memory).

Figure 7 compares, in the case of the batched matrix inversion kernel presented in Section III-C2, our strategy of aggressively utilizing shared memory (first bar) to the one that still exploits all parallelism, but utilizes only global memory, with all accesses being coalesced (second bar). Several datasets are redundant because the number of flops $6MK^3$ is invariant to `N` and `n`. Our strategy generates a $5-6\times$ speed-up, and offers stable performance around 400 GFlops^{Sp}. This number may seem small, but this is due to the arithmetical intensity of the implementation being low—the number of flops per memory accesses is under one. In fact the Gbytes/sec^{Sp} performance is about $3.5\times$ the peak bandwidth of the system.

C. Application-Level Performance

The overall performance is reported in Figure 8. The bars denote, in order, several implementations generated under the different optimization strategies discussed in Section III-B:

Ours uses register tiling and aggressively utilizes inner-parallelism in fast (shared) memory. The performance is stable at about 950GFlops^{Sp} for all datasets but `D6`, where it drops to 575GFlops^{Sp}. Reasons are (i) the inefficient-transposition issue discussed in the previous section, and (ii) the fact that reducing `n` from `N/2` to `N/4` diminishes the weight of the efficiently-tiled computations in the total runtime.

RgTl-EfSeq denotes a version in which matrix multiplication-like computations have been efficiently tiled, but the inner parallelism of the remaining code has been efficiently sequentialized. The figure shows that the impact of utilizing inner-parallelism in fast memory is a factor between $2\times$ and $3\times$ (on `D5`), except for `D6` and `Africa(Small)`, where the higher weight of missing values narrows the performance gap.

Full-EfSeq denotes the "naive" strategy that fuses everything into one kernel and exploits only the outer parallelism of size `M` (of the pixels in the image). The performance gap between **RgTl-EfSeq** and **Full-EfSeq** demonstrates the impact of tiling matrix multiplication-like kernels, which results in $1.5 - 2\times$ speed-up at application level.

C denotes a hand written parallel `C` implementation based on OpenMP, which is decently optimized for locality of reference,[10] but vectorization is left to what the GCC compiler can extract. The parallel-CPU speed-up is about $21\times$, which is close to optimal given that it was obtained on a 16-core CPU with 2-way hyperthreading.

The **Ours** GPU version outperforms the OpenMP-parallel **C** by a factor between $24\times$ on `D6` to $48\times$ on `D5`, which validates the usefulness of the GPU acceleration for BFAST-Monitor.

V. Large-Scale Change Detection

Next, we will demonstrate the applicability of the overall framework in the context of large-scale change detection scenarios. We would like to stress that the purpose of the following experiments is *not* to compare different change detection methods: Our parallel implementation of BFAST-Monitor yields the same results as the commonly used `R` implementation (up to machine precision) and we do not claim any algorithmic/methodical improvements over the BFAST-Monitor approach. For a methodical comparison of different change detection methods with BFAST-Monitor, we therefore refer the reader to the related work [3], [30], [31]. Our

[10]Please note that high-performance libraries, such as Intel's MKL were not used for two reasons. First, the (filtered) matrix multiplication under variant mask—presented in section III-C1—is not supported by such libraries. Second, in our implementation each thread reuses the memory necessary for executing one iteration, thus maximizing the cache performance. Using MKL to implement the batch operations would require distributing the computation such that matrix multiplication/inversion is performed for all (or a chunk of) pixels, prior to performing the rest of the computation. This would require a much-bigger memory footprint, which would likely result in poorer locality.

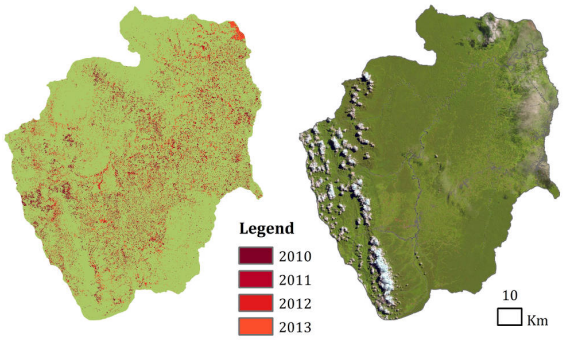


Fig. 9: `Peru(Large)`: Detected changes (left) as well as image data (RGB) for one time step (right).

work focuses on the computational performance and it is very difficult to compare in this respect with other methods implementations in this field, since they are all implemented in high-level programming languages such as `Python` or `R` and are not not optimized from a compute perspective [31], see again Section II-C. Also note that the only existing GPU implementation for BFAST-Monitor cannot be applied since it addresses a simpler scenario, in which the time series do not contain any missing values [27].

A. Experimental Setup

To show the practical benefits of our novel GPU implementation, we resorted to a standard desktop system with an `Intel(R) Core(TM) i5-8600K` CPU at 3.60GHz, 32GB RAM, and a `GeForce GTX TITAN Z` GPU with 2880 shader units (6GB RAM), and considered large datasets with up to billions of individual pixels to be processed. The results were compared with the ones obtained via the available `R` implementation executed on the same system using all 6 CPU cores (parallel execution over the pixels). We tested our implementation on three real-world datasets of increasing size, which were acquired through the *Google Earth Engine* [36] and which are based on data from the Landsat [22] program:[11]

- *`Peru(Small)`:* The smallest dataset is based on a 10x10km area in the south of the Loreto region in Peru, where the main driver for deforestation was agricultural expansion, see again Figure 3. The dataset is based on image data with $M = 334 \cdot 334$ pixels from $N = 216$ dates between 01/01/2000 and 31/12/2016.
- *`Peru(Large)`:* The second dataset is based on the entire province of Padre Abad in the central Amazon rainforest of Peru, see Figure 9 for an illustration. This area of over 8800 km^2 is one of the most affected regions in Peru regarding deforestation. The dataset contains 16GB of images with $M = 4458 \cdot 3678$ pixels from $N = 488$ dates (01/01/2000 to 31/12/2013).
- *`Africa`:* The third dataset is based on the entire continental tropical Africa (between 20°N and 20°S latitude). The dataset contains 38234 images (2167GB) with $M = 221 \cdot 768$ pixels based on 6873 dates between 01/01/2000 and 31/12/2018. Note that for each individual image, one is given only about $N = 350$ slices that contain any data (these slices are extracted efficiently before applying the core algorithm, see Section III-D).[12] The `Africa(Small)` dataset considered in the previous section contains the data from one of the images and is based on $N = 327$ dates/slices.

We investigate both the runtime of the different phases as well as the overall runtimes needed to process the entire scenarios.

B. Practical Runtime Analysis

The practical runtimes of our implementation (based on the **Ours** version) on the three datasets are shown in Figure 10. The single images of both `Peru(Large)` and `Africa` were too big to fit into GPU memory and were, hence, split into 50 chunks. We measured the preprocessing time (host), the transfer time (between host and GPU), the kernel execution time (GPU), as well as the overhead for the chunking (host), see Section III-D. The runtimes reported depict averages over ten runs for both `Peru(Small)` and `Peru(Large)`. For `Africa`, the average runtime *per single image* is reported (computed based on 50 random images).

It can be seen that (1) the transfer time between host and GPU is generally smaller than the one for the execution of the kernel and that (2) the preprocessing time and the time needed to partition the data into chunks are, together, close to the kernel execution time. Thus, the computations for these phases can be interleaved, leading to the kernel execution time dominating the overall runtime (thus, the runtimes allow a direct comparison with the `C` implementation that was analyzed in the previous section). Note that the `R` implementation needed more than 25 hours to process `Peru(Large)` (using all 6 CPU cores), whereas the GPU implementation could process this scenario in about 17 seconds, which corresponds to a speed-up of more than $5000\times$.[13] It is worth noting that loading the images from disk to host and decompressing them has become the new bottleneck (e.g., about 25 seconds for `Peru(Large)`). However, from a practical perspective, this is not a problem since many BFAST-Monitor runs are generally conducted (e.g., for different history/monitoring periods or for different values for k), which compensates this overhead.

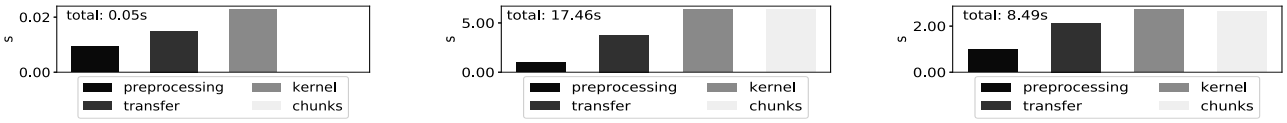
C. Unsupervised Change Detection for Massive Data

Finally, we conducted large-scale experiments for all datasets, where we varied the start and the end of the monitoring periods. More precisely, different periods starting at 2010 were considered, each lasting one year (i.e., 2010–2011, 2011-2012, ...). In addition to the break indices (and the associated

[11]Collection 1 Tier 1 Surface Reflectance derived NDMI images of different Landsat 5, 7, and 8 sensors. Data was only collected for the pixels that were considered to be forest in 2010 (global tree cover with threshold 30 [37]).

[12]For areas extending over multiple satellite swaths (data from adjacent Landsat swaths are acquired on different days), each image is often extended with missing values (NaN) to create one data-cube for the entire area.

[13]The main reason for this significant speed-up is the fact that the baseline implementation resorts to the high-level programming language `R` and that is also not optimized for parallel execution.



(a) `Peru(Small)` (b) `Peru(Large)` (c) `Africa`

Fig. 10: Practical runtime in seconds (s) needed by our GPU implementation for the three different datasets. For `Africa`, the average runtime needed to process a single image is reported (processing all 38234 images takes about 90 hours).

time), BFAST-Monitor outputs the magnitude of the change for the monitoring period (i.e., the mean of the MOSUM process). A positive mean can be interpreted as an increase in greening, or moisture, which are associated with an increase in vegetation, and vice-versa. The results for `Peru(Small)`, `Peru(Large)`, and for `Africa` are shown in Figure 3, Figure 9, and Figure 11, respectively. In each case, the figure illustrates where and when a break with negative magnitude occurred, which can be interpreted as vegetation decrease after 2010 (we refer to the literature in remote sensing [4]–[8], [15], [17]–[21] for a more detailed discussion of such results).

Using our GPU implementation, it takes about 90 hours to process a single monitoring period for `Africa` (e.g., 2010–2011), and about four weeks to process the whole `Africa` scenario using a single GPU.[14] Note that scenarios of this size have never been analyzed before as it would take years to obtain the results with the available `R` implementation, even in case multiple powerful multi-core machines were used.

Acknowledgements: FG and CO would like to acknowledge support from the Danish Industry Foundation through the IDAS project and from the Independent Research Fund Denmark (DFF) through the grant *Monitoring Changes in Big Satellite Data via Massively-Parallel Artificial Intelligence*. CO would also like to acknowledge support from DFF through the project *FUTHARK: Functional Technology for High-performance Architectures*. SR and JV would like to acknowledge support from Food and Agriculture Organization of the United Nations (FAO-UN) through the framework of the FAO project TF/FOMDD/TF5C350011682 SEPAL.

[14]To obtain the results for `Africa`, a cluster with 20 GPUs was used.

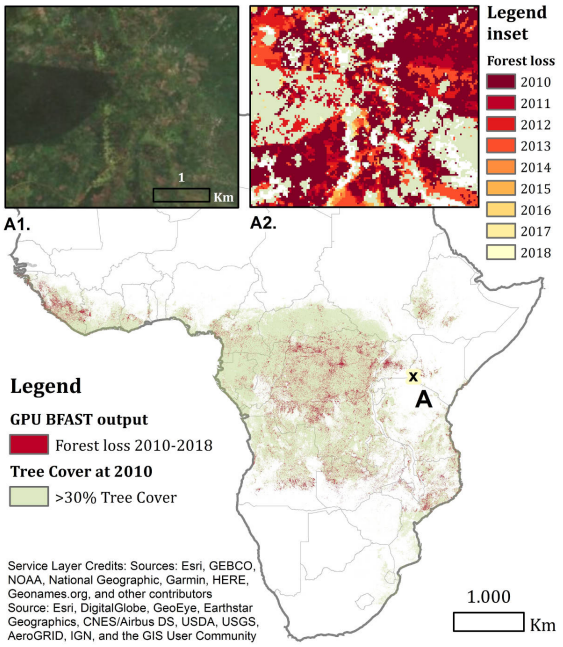


Fig. 11: Large-scale GPU BFAST application over the entire continental tropical Africa. The analysis was performed only over pixels with more than 30% tree cover in 2010 [37]. The image illustrates the detected breaks with a negative magnitude (red). The inset figures zoom in on a deforested area. A1 emphasizes the timing of the detected breaks and A2 shows the deforestation on RGB satellite imagery.

References

[1] T. Pistorius, "From red to redd+: the evolution of a forest-based mitigation approach for developing countries," *Current Opinion in Environmental Sustainability*, vol. 4, no. 6, pp. 638–645, 2012.

[2] M. Herold and M. Skutsch, "Monitoring, reporting and verification for national redd+ programmes: two proposals," *Environmental Research Letters*, vol. 6, no. 1, p. 014002, 2011.

[3] Z. Zhu, "Change detection using landsat time series: A review of frequencies, preprocessing, algorithms, and applications," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 130, pp. 370–384, 2017.

[4] B. DeVries, M. Decuyper, J. Verbesselt, A. Zeileis, M. Herold, and S. Joseph, "Tracking disturbance-regrowth dynamics in tropical forests using structural change detection and landsat time series," *Remote Sensing of Environment*, vol. 169, pp. 320 – 334, 2015.

[5] E. Hamunyela, J. Verbesselt, and M. Herold, "Using spatial context to improve early detection of deforestation from landsat time series," *Remote Sensing of Environment*, vol. 172, pp. 126–138, 2016.

[6] J. Reiche, R. Lucas, A. L. Mitchell, J. Verbesselt, D. H. Hoekman, J. Haarpaintner, J. M. Kellndorfer, A. Rosenqvist, E. A. Lehmann, C. E. Woodcock, F. M. Seifert, and M. Herold, "Combining satellite data for better tropical forest monitoring," *Nature Climate Change*, vol. 6, pp. 120–122, 2016.

[7] M. Romero-Sanchez and R. Ponce-Hernandez, "Assessing and monitoring forest degradation in a deciduous tropical forest in mexico via remote sensing indicators," *Forests*, vol. 8, no. 9, p. 302, 2017.

[8] P. Murillo-Sandoval, J. Van Den Hoek, and T. Hilker, "Leveraging multi-sensor time series datasets to map short-and long-term tropical forest disturbances in the colombian andes," *Remote Sensing*, vol. 9, no. 2, p. 179, 2017.

[9] R. de Jong, J. Verbesselt, A. Zeileis, and M. Schaepman, "Shifts in global vegetation activity trends," *Remote Sensing*, vol. 5, no. 3, pp. 1117–1133, Mar. 2013.

[10] R. Jong, J. Verbesselt, M. E. Schaepman, and S. Bruin, "Trend changes in global greening and browning: contribution of short-term trends to longer-term change," *Global Change Biology*, vol. 18, no. 2, pp. 642–655, 2012.

[11] S. Horion, A. V. Prishchepov, J. Verbesselt, K. de Beurs, T. Tagesson, and R. Fensholt, "Revealing turning points in ecosystem functioning over the northern eurasian agricultural frontier," *Global Change Biology*, vol. 22, no. 8, pp. 2801–2817, Aug. 2016.

[12] S. Hislop, S. Jones, M. Soto-Berelov, A. Skidmore, and others, "A fusion approach to forest disturbance mapping using time series ensemble techniques," *Remote Sens. Environ.*, 2019.

[13] C. Abel, S. Horion, T. Tagesson, M. Brandt, and R. Fensholt, "Towards

improved remote sensing based monitoring of dryland ecosystem functioning using sequential linear regression slopes (SeRGS)," *Remote Sens. Environ.*, vol. 224, pp. 317–332, Apr. 2019.

[14] E. L. Bullock, C. E. Woodcock, and C. E. Holden, "Improved change monitoring using an ensemble of time series algorithms," *Remote Sens. Environ.*, p. 111165, May 2019.

[15] V. Smith, C. Portillo-Quintero, A. Sanchez-Azofeifa, and J. L. Hernandez-Stefanoni, "Assessing the accuracy of detected breaks in landsat time series as predictors of small scale deforestation in tropical dry forests of mexico and costa rica," *Remote Sens. Environ.*, vol. 221, pp. 707–721, Feb. 2019.

[16] J. Verbesselt, A. Zeileis, and M. Herold, "Near real-time disturbance detection using satellite image time series," *Remote Sensing of Environment*, vol. 123, pp. 98 – 108, 2012.

[17] N. Tsutsumida, I. Saizen, M. Matsuoka, and R. Ishii, "Land cover change detection in ulaanbaatar using the breaks for additive seasonal and trend method," *Land*, vol. 2, no. 4, pp. 534–549, 2013.

[18] X. Cai, L. Feng, X. Hou, and X. Chen, "Remote sensing of the water storage dynamics of large lakes and reservoirs in the yangtze river basin from 2000 to 2014," *Scientific reports*, vol. 6, p. 36405, 2016.

[19] X. Che, Y. Yang, M. Feng, T. Xiao, S. Huang, Y. Xiang, and Z. Chen, "Mapping extent dynamics of small lakes using downscaling modis surface reflectance," *Remote Sensing*, vol. 9, no. 1, p. 82, 2017.

[20] J. Liu, J. Heiskanen, E. E. Maeda, and P. K. Pellikka, "Burned area detection based on landsat time series in savannas of southern burkina faso," *International journal of applied earth observation and geoinformation*, vol. 64, pp. 210–220, 2018.

[21] M. Urbazaev, C. Thiel, M. Migliavacca, M. Reichstein, P. Rodriguez-Veiga, and C. Schmullius, "Improved multi-sensor satellite-based aboveground biomass estimation by selecting temporally stable forest inventory plots using NDVI time series," *Forests*, vol. 7, no. 8, p. 169, 2016.

[22] M. A. Wulder, J. G. Masek, W. B. Cohen, T. R. Loveland, and C. E. Woodcock, "Opening the archive: How free data has enabled the science and monitoring promise of landsat," *Remote Sensing of Environment*, vol. 122, no. Supplement C, pp. 2 – 10, 2012, landsat Legacy Special Issue.

[23] J. Li and D. P. Roy, "A global analysis of sentinel-2a, sentinel-2b and landsat-8 data revisit intervals and implications for terrestrial monitoring," *Remote Sensing*, vol. 9, no. 902, 2017.

[24] M. Schultz, J. G. Clevers, S. Carter, J. Verbesselt, V. Avitabile, H. V. Quang, and M. Herold, "Performance of vegetation indices from landsat time series in deforestation monitoring," *Int. Journal of Applied Earth Observation and Geoinformation*, vol. 52, pp. 318–327, 2016.

[25] A. Zeileis, A. Shah, and I. Patnaik, "Testing, monitoring, and dating structural changes in exchange rate regimes," *Comput. Stat. Data Anal.*, vol. 54, no. 6, pp. 1696–1706, Jun. 2010.

[26] M. Reichstein, G. Camps-Valls, B. Stevens, M. Jung, J. Denzler, N. Carvalhais, and Prabhat, "Deep learning and process understanding for data-driven Earth system science," *Nature*, vol. 566, no. 7743, pp. 195–204, 2019.

[27] M. von Mehren, F. Gieseke, J. Verbesselt, S. Rosca, S. Horion, and A. Zeileis, "Massively-parallel break detection for satellite data," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*. ACM, 2018, pp. 5:1–5:10.

[28] T. Henriksen, F. Thorøe, M. Elsman, and C. E. Oancea, "Incremental flattening for nested data parallelism," in *Procs. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2019.

[29] C. C. Aggarwal, *Outlier Analysis*, 2nd ed. Springer, 2016.

[30] A. Ben Abbes, O. Bounouh, I. R. Farah, R. de Jong, and B. Martínez, "Comparative study of three satellite image time-series decomposition methods for vegetation change detection," *European Journal of Remote Sensing*, vol. 51, no. 1, pp. 607–615, Jan. 2018.

[31] K. Awty-Carroll, P. Bunting, A. Hardy, and G. Bell, "An evaluation and comparison of four dense time series change detection methods using simulated data," *Remote Sensing*, vol. 11, no. 23, p. 2779, 2019.

[32] G. E. Blelloch, "Scans as Primitive Parallel Operations," *Computers, IEEE Transactions*, vol. 38, no. 11, pp. 1526–1538, 1989.

[33] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. John Wiley & Sons, 2014.

[34] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates," in *Procs. ACM SIGPLAN Conf. on Prog. Lang. Design and Implem. (PLDI)*, 2017, pp. 556–571.

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

[36] N. Gorelick, M. Hancher, M. Dixon, S. Ilyushchenko, D. Thau, and R. Moore, "Google earth engine: Planetary-scale geospatial analysis for everyone," *Remote Sensing of Environment*, 2017.

[37] M. Hansen, P. Potapov, R. Moore, M. Hancher, S. Turubanova, A. Tyukavina, D. Thau, S. Stehman, S. Goetz, T. Loveland, A. Kommareddy, A. Egorov, L. Chini, C. Justice, and J. Townshend, "High-resolution global maps of 21st-century forest cover change," *Science (New York, N.Y.)*, vol. 342, pp. 850–853, 11 2013.

APPENDIX

1) Notation: We use **`i32`** and **`f32`** to denote 32-bits integer and floating-point types, respectively. We use `[N][M]`**`f32`** as the type of a two-dimensional array of floats with `N` rows and `M` columns. We use `[`a_1`,...,`a_n`]` to denote an array literal and `(0,...,n-1)` to denote an array containing the integers from `0` to `n-1`. We use for example `(`**`i32`**`,[N]`**`f32`**`,[N]`**`f32`**`)` to denote a tuple (record) type containing an integer, and two length-`N` arrays of floats. Similarly, we use `(a,b)` to denote a tuple value. Here are the semantics of some operators:

$$\textbf{map}\ f\ [a_1,\ldots,a_n] \ =\ [f\ a_1,\ldots,f\ a_n]$$
$$\textbf{map2}\ f\ [a_1,\ldots,a_n][b_1,\ldots,b_n] \ =\ [f\ a_1\ b_1,\ldots,f\ a_n\ b_n]$$
$$\textbf{reduce}\ \oplus\ 0_\oplus\ [a_1,\ldots,a_n] \ =\ 0_\oplus \oplus a_1 \oplus \ldots \oplus a_n$$
$$\textbf{scan}\ \oplus\ 0_\oplus\ [a_1,\ldots,a_n] \ =\ [a_1, a_1 \oplus a_2, \ldots, a_1 \oplus \ldots \oplus a_n]$$

Map produces a result array by applying its function argument `f` to each element of its input array. The function can be declared in the program or can be an anonymous (lambda) function; for example **`map`** `(λx→x+1) arr` adds one to each element of array `arr`. Similarly, **`map2`** applies its function argument to (corresponding) elements from its two array parameters. Please note that calling function `f` on two arguments `a` and `b` is written as `f a b`, i.e., without any parenthesis or commas. Reduce successively applies a binary-associative operator ($\oplus$) to the elements of its input array, and can be parallelized by a reduction-tree computation ($0_\oplus$ is the neutral element of $\oplus$). Scan [32] is similar to reduce, except that it produces an array of length `n` containing all prefix sums of its input array. In addition, **`replicate`** `n a` creates an array of length `n` whose elements are all `a`. Finally, **`scatter`** `x is vs` updates in place the array `x` at indices contained in array `is` with the values contained in array `vs`, but out-of-bounds indices are ignored (not updated). For example, **`scatter`** `[0.5,1.0,1.5] [2,-1,0] [3.5,.5,2.5]` results in array `[2.5,1.0,3.5]`. We use `|>` to pipe the result of a function application as input to another function, for example **`map`** `(λx→x+1) arr |>` **`reduce`** `(+) 0` adds one to each element of `arr` and then sums up the result array.

Finally, a function declaration consists of a sequence of typed parameters of form `(name: type)`, optionally a result type, and an expression (the function body). An expression can be a function call, binary operation, comparison, an **`if`** `cond` **`then`** `e1` **`else`** `e2` expression—which has similar semantics to the C expression `(cond ? e1 : e2)`—or a **`let`** expression. The latter can be seen as a sequence of (**`let`**) statements followed by one return denoted by the **`in`** keyword.

2) Details: The data-parallel specification is presented in Figure 12, where the entry-point function is named `bfast`. For the most part we kept the names consistent with those in section III-A, except that we use $\mathbf{X}^{sqr}$ and $\mathbf{X}^{sqr^{-1}}$ instead of $\overline{\mathbf{M}}$ and $\overline{\mathbf{M}}^{-1}$. For example, $\mathbf{Y}$`:[M][N]f32` denotes an image with `M` pixels and a time-series of `N` values per pixel, and `hf` denotes the fraction of the valid values in the training set that give the width of the MOSUM window. The result is a length-`M` array of tuples, recording the index of the first break (or `-1` if none) and the MOSUM mean for each pixel. A preliminary step computes (once) the matrix $\mathbf{X}$ by parallel function `mkX`, its transpose $\mathbf{X}^T$, and their slices $\mathbf{X}_{[:,:n]}$ and $\mathbf{X}^T_{[:,:n]}$. The main computation consists of the outer **`map`** which applies the anonymous function implementing BFAST-Monitor to the time series $\mathbf{y}$ of each pixel of the image: First, $\mathbf{X}_{[:,:n]}$ is squared under the mask of $\mathbf{y}_{[:n]}$, i.e., the matrix multiplication ignores the elements in the rows of $\mathbf{X}_{[:,:n]}$ for which the corresponding value in $\mathbf{y}_{[:n]}$ is invalid (`NaN`). Then the obtained matrix is inverted by Gauss-Jordan elimination, and the result is recorded in $\mathbf{X}^{sqr^{-1}}$. These steps—i.e., functions `mmMulFilt` and `matInv`—are discussed in sections III-C1 and III-C2. Second, the matrix $\mathbf{X}_{[:,:n]}$ is multiplied with the vector $\mathbf{y}_{[:n]}$ under the mask $\mathbf{y}_{[:n]}$, and the resulting vector is multiplied with the matrix $\mathbf{X}^{sqr^{-1}}$ to produce $\overline{\boldsymbol{\beta}}$. The implementation of `mvMulFilt` is shown at the top of Figure 12, and the one of `mvMul` is the standard matrix-vector multiplication—i.e., `1.0-(isnan y)` is missing. The final matrix-vector multiplication between $\mathbf{X}^T$ and $\overline{\boldsymbol{\beta}}$ yields the prediction $\hat{\mathbf{y}}$ for the whole time series. Third, the error between actual values and prediction is computed (**`map2`** `(-) y` $\hat{\mathbf{y}}$) and the invalid (`NaN`) values are filtered out from the result. This is accomplished by the function `filterNaNsWKeys` resulting in $\overline{N}$—the number of valid values—together with arrays $\overline{\mathbf{r}}$ and $\overline{\mathbf{I}}$, which record consecutively, the prediction error of the valid values and their indices in the original time series, respectively.[15]

In the remaining code, the padding is not shown to simplify the notation. The number of valid entries in the training set $\overline{n}$, the MOSUM process parameter $\hat{\sigma}$, and the first value of the MOSUM process m^{fst} are computed each by a map-reduce operation. Then the whole MOSUM process $\overline{\mathbf{m}}$ is computed by a **`map`**-**`scan`**-**`map`** composition, and its `mean` is computed by a reduction. Lastly, the index of the first break `brk` is computed by a (**`map`**-)**`reduce`** operating on boolean-integer tuples recording whether a break was found at the current index. If it exists, the index of the first break is remapped to the one in the original time series, otherwise `-1` is reported. The comments `-- ker i` delimit a computation that is a composition of operations of same-parallel size, which will be mapped to a CUDA kernel i, as explained in Section III-B.

[15]The implementation of `filterNaNsWKeys`, shown in Figure 12, first marks in `tfs` the valid/invalid entries with one/zero, then it scans `tfs` to obtain the consecutively-reordered indices for the valid entries. The following **`map2`** turns the invalid indices to `-1`, so that they are ignored by the final two **`scatter`** operations that reorder the valid values and their original time-series indices in arrays `vs` and `ks`, respectively. The logical length of `vs` and `ks` is the last element of scanned array `indsT`. The arrays are padded to length `N` because the logical length varies across pixels, leading otherwise to irregular computation across pixels and expensive bookkeeping overhead.

```
let mvMulFilt[n][m](xss:[n][m]f32)(ys:[m]f32)=
  map(λxs → map2(λx y → x * y * (1.0-(isnan y))
                ) xs ys |> reduce (+) 0
     ) xss
let filterNaNsWKeys [N] (vec: [N]f32) :
                    (i32, [N]f32, [N]i32) =
  let tfs = map (λv → 1 - (isnan v)) vec
  let indsT = scan (+) 0 tfs
  let inds = map2 (λi tf → tf*i-1) indsT tfs
  let vs=scatter (replicate N NAN) inds vec
  let ks=scatter (replicate N 0) inds (0...N-1)
  in  (indsT[N-1], vs, ks)

entry bfast[M][N] (k:i32) (n:i32) (f:f32)(λ:f32)
       (hf: f32)(Y: [M][N]f32): [M](i32,f32) =
 let K  = 2*k + 2
 let X = mkX K f -- [K][N]f32            -- ker 1
 let X^T = transpose X
 let (X_[:,:n], X^T_[:,:n]) = (X[:,:n],X^T[:n,:]) in
 map(λy → let y_[:n] = y[:n]
       -- X^sqr, X^sqr^-1 :[K][K]f32; β_0, β :[K]f32
       let X^sqr = mmMulFilt X_[:,:n] X^T_[:,:n] y_[:n]  -- ker 2
       let X^sqr^-1 = matInv X^sqr              -- ker 3
       let β_0  = mvMulFilt X_[:,:n] y_[:n]       -- ker 4
       let β    = mvMul X^sqr^-1 β_0              -- ker 5
       -- ŷ, r̄, Ī : [N]f32
       let ŷ  = mvMul X^T β                      -- ker 6
       let (N̄,r̄,Ī)= map2 (-) y ŷ |>              -- ker 7
                    filterNaNsWKeys -- ker 7
       -- ker 8; inner-parallel size: n̄
       let n̄= map (λv → 1 - (isnan v)) y_[:n]
            |> reduce (+) 0
       let σ̂_0 = map (λa → a*a) (r̄[:n̄])
            |> reduce (+) 0.0
       let σ̂ = sqrt (σ̂_0 / (r32 (n̄-K)))
       let h = t32 ((r32 n̄) * hf)
       -- ker 9; inner-parallel size: h
       let m^fst= map(λi → r̄[i+n̄-h+1]) (0...h-1)
            |> reduce (+) 0.0
       -- ker 10; inner-parallel size: N̄-n̄
       let m̄^0=map(λt→ if t==0 then m^fst
                       else r̄[n̄+t]-r̄[n̄-h+t]
                  )(0...N̄-n̄-1) |> scan (+) 0.0
       let m̄ = map (λm^0_t → m^0_t/(σ̂*(sqrt (r32 n̄)))
                   ) m̄^0
       let mean = reduce (+) 0.0 m̄
       let (found_break, brk) =
         map2 (λm_t t →
                  let b_t = λ * (sqrt (log+(t/n̄)))
                  in  ((abs m_t) > b_t, t)
              ) m̄ (0...N̄-n̄-1) |>
         reduce ( λ(b1,i1) (b2,i2) →
                  if b1 then(b1,i1) else(b2,i2)
                ) (false,-1)
       let brk'= if !found_break then -1
                 else remapIndices brk n n̄ N̄ Ī
       in (brk', mean)
    ) Y
```

Fig. 12: Functions `r32` and `t32` implement conversions between real and integral values. Function `isnan` returns either integer or real 0 if the argument is a `NAN` value, or 1 if it is not. Function `mvMul` denotes matrix-vector multiplication and it is similar to `mvMulFilt` except that factor `1-(isnan y)` is missing. Functions `mmMulFilt` and `matInv` are discussed in Sections III-C1 and III-C2.