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

Anonymous #1, Anonymous #2, Anonymous #2, Anonymous #3, Anonymous #3

***Abstract*—Mitigating climate change is one of the greatest challenges that threatens the future of humanity of our time. After increasing use of fossil fuels, deforestation is the second largest source of greenhouse gas emissions and monitoring such environmental changes is an essential component of the fight against climate change. Over the last decades, large amounts of satellite data have been collected, which, in combination with appropriate methods, offer the opportunity to derive precise, valuable estimates for deforestation. While various change detection techniques have been proposed, the associated implementations do not scale well to large amounts of data. In this work, we propose a novel massively-parallel implementation of a popular detection scheme called BFAST, which can handle huge amounts of data efficiently. We demonstrate the potential of our implementation in the context of large-scale deforestation detection tasks. Instead of months or even years, such scenarios can now be handled in a few days using single desktop systems.**

I. Introduction

In the context of climate change, estimating and reducing global deforestation is a crucial for strategy to mitigate the effect of global warming [1]. This renders mapping forest change at large scales more important than ever before [2]. Remote sensing is considered an essential, objective, and cost-effective way for mapping forests. In particular, time-series analysis of satellite data has been established as a tool for monitoring forest change [3], see Figure 1.

One of the state-of-the-art methods in this context is the *break detection for additive season and trend* (BFAST) approach, which generates, for each pixel, an additive season and trend model [4]. While this method and related approaches have been applied to a variety of important problems including monitoring deforestation [5]–[7], the induced practical runtime usually becomes a major bottleneck and has, so far, limited their application to relatively small areas. The Landsat [8] and Sentinel [9] catalogs nowadays produce petabytes of data. For such catalogs, deforestation monitoring at continental scales requires, so far, immense computational resources. Recent work has demonstrated the potential of high-performance computing for accelerating BFAST [10], albeit in a simpler scenario, which assumed that all time-series values are valid. Unfortunately, this restriction makes it impractical for many vegetated areas on Earth, where clouds frequently mask image pixels. This work addresses the general case. In particular, we provide a massively-parallel implementation for BFAST that can effectively handle scenarios in which the input time series contain many missing values. From a computational perspective, this case is considerably harder to parallelize, since the individual tasks are very *irregular* in the sense that one is given very different time series for the individual pixels.

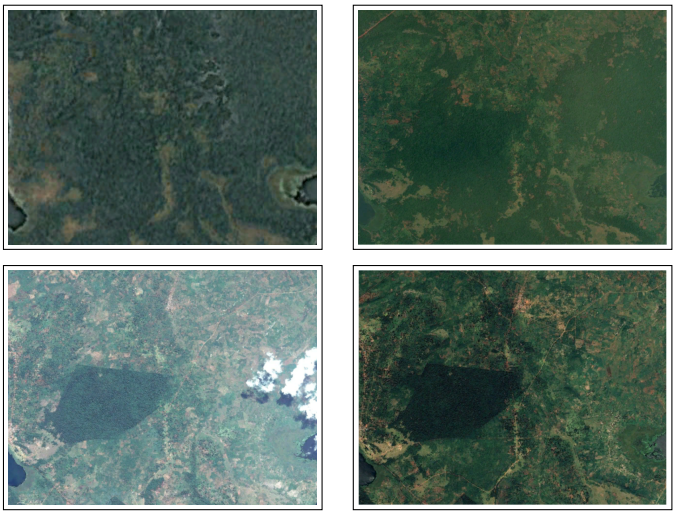


Fig. 1: Deforestation visible in a sequence of satellite images obtained in 2002, 2010, 2014, and 2017. Typically, sequences with hundreds of images are considered, which yield, for each individual pixel, a time series of reflectance measurements.

II. Background and Contributions

A. Unsupervised Change Detection for Satellite Data

The *BFAST (Monitor)* method [4] consists in fitting a linear regression model on a period defined as *stable history period* and by testing for stability of the same model during a period defined as *monitoring period*. The mean deviations of the monitored values are provided as *change magnitude* values, and if deviation is significant at a specific moment in time, a *break* is recorded. BFAST is a generic time series technique that detects change from pre-processed satellite data. For detecting changes in forest cover, wetness related spectral indices such as the *Normalized Difference Moisture Index (NDMI)* are used, as they are simple and robust techniques to extract quantitative information on the amount of vegetation for every pixel in an image [11]. Thus, given multiple such images for the same region of interest, a time series $y_1, \ldots, y_N$ is created for *each* pixel, see Figure 2. BFAST assumes, for each pixel, a model of the form:

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

Here, the first and second term determine the intercept and the trend, respectively. The third term specifies the seasons.

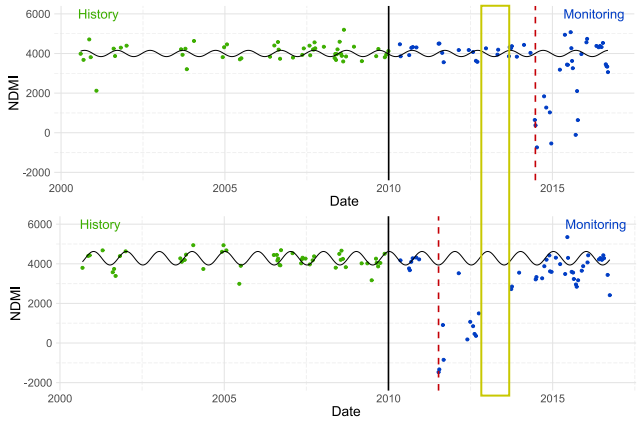


Fig. 2: The figure shows two BFAST examples [4]. For each time series, a model is derived based on data given for a history period (green points). In case the predictions generated by this model are not in line with the data available for the monitoring phase (blue points), a "break" is detected (red dashed line). Each such time series corresponds to a pixel of a series of satellite images. The bands of each such multi-spectral satellite image are converted to a single index (here, the NDMI index). Note that the individual time series usually vary both w.r.t. the number and location of points (yellow rectangle).

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 365 observations per year or $f = 23$ for time series with an interval of 16 days between the different observations. 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 $t = 1, \ldots, N$. The model (1) can be written as a standard linear model of the form

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

with non-linear patterns

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

that are generated based on each date t, where $F_t(j) = {}^{2\pi jt}/_{f}$.[1]

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

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

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 (where b_t is specified by the user) and no break should be detected. Otherwise, a break is detected, see 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 $K = 2 + 2k$ that contains, for each date t, the constructed pattern $\mathbf{x}_t$ defined in Equation (3) along with the pixel value y_t (e.g., the NDMI index).[2] Afterwards, a solution for

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

is computed, which can be used to obtain the MOSUM process (4) as well as the break points for the monitoring phase. 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. Note that, typically, not all measurements y_i are available for a certain pixel due to, e.g., clouds in the satellite data (which are masked out). These values correspond to missing data, which are filtered out before fitting the model and before computing the MOSUM process. Hence, the dataset T typically varies from pixel to pixel, yielding many *different* individual regression problems.

B. Contribution

This work proposes a massively-parallel implementation that can address general change detection scenarios with potentially many missing values per time series (which stem from, e.g., clouds in the satellite images). This gives rise to a massive amount of individual learning tasks with both different computational demands and different memory access patterns.

Previous work addressed the special case, where the missing values occurred for the same dates t for *all* pixels, which leads to datasets T having the same length and sharing the same patterns $\mathbf{x}_i$ (which is often not the case, see the highlighted region in Figure 2). This regularity could be exploited for the computation of the individual models [10]. More precisely, the inverted squared data matrix (see below) is the same for all pixels and has to be computed only once for *all* the pixels (which also leads to a significantly smaller asymptotic work complexity). Our approach is fundamentally different and can handle the general and much more common case. In particular, we make the following contributions:

- We present a high-level specification of our parallel algorithm that describes all available (nested) parallelism.

[1] It can be shown that the optimal coefficients are of the form $\boldsymbol{\beta}^* = \{\alpha_1, \alpha_2, \gamma_1 \cos(\delta_1), \gamma_1 \sin(\delta_1), ..., \gamma_k \cos(\delta_k), \gamma_k \sin(\delta_k)\}^\top$, which relates the coefficients computed to both the amplitudes γ_i and phases δ_i.

[2] In practice, changes are computed based on the assumption that no breaks occurred during the history period for the individual pixels. This is a typical scenario for the monitoring of, e.g., deforestation, where the mean of the deviations can also be used to check if deforestation or reafforestation happened in a certain area. Thus, the change detection can be conducted in an unsupervised fashion, which depicts a big advantage compared to supervised deep learning methods that resort to labels for the individual changes.

We also report a fully-automated compilation strategy that effectively maps the specified parallelism to graphics processing units (GPU).
- 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.
- Our implementation is two to three orders of magnitudes faster than the commonly used BFAST implementation (implemented in `R`). It allows researchers, for the first time, to handle massive change detection scenarios covering large parts of our Earth in a reasonable amount of time using single standard desktop computers. In addition to the performance analysis of the GPU kernels, we conducted a large-scale experiment covering the entire continental tropical Africa—a scenario that was basically impossible to handle before.

Our implementation—which will be made publicly available—dramatically reduces the runtime needed for monitoring changes and we expect that it will play an important role for future analyses conducted in remote sensing.

III. Algorithmic Framework

A. Detecting Breaks in Time Series with Missing Values

The algorithmic building blocks of BFAST are given 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 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. Both $\overline{\mathbf{y}}$ and $\overline{\mathbf{X}}$ are used to fit a standard linear regression model $\overline{\boldsymbol{\beta}} = \overline{\mathbf{M}}^{-1}\overline{\mathbf{X}}_{[:,:\overline{n}]}\overline{\mathbf{y}}_{[:\overline{n}]}$ for the history period, where $\overline{n}$ specifies the new end of the history period in $\overline{\mathbf{X}}$. 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 $\mathbf{y}$ (which 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).

B. Design Space of Parallel Implementations at a High-Level

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 at Lines 12 and 13 are also trivially parallel. Finally, the filtering operation at 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 [12]. A further complication is that the inner-parallel operations have *different sizes*, but the GPU hardware provides morally a flat-parallel programming interface; it follows that significant rewriting is necessary to map that parallelism to a form that the GPU can exploit.

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 fuse more aggressively 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 [12], such that it can be mapped

Algorithm 1 BFAST

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: $\overline{\mathbf{M}} = \text{Multipy}(\overline{\mathbf{X}}_{[:,:\overline{n}]}, (\overline{\mathbf{X}}_{[:,:\overline{n}]})^\top)$ ▷ $\overline{\mathbf{M}} \in \mathbb{R}^{K\times K}$
3: $\overline{\mathbf{M}}^{-1} = \text{InvertMatrix}(\overline{\mathbf{M}})$ ▷ $\overline{\mathbf{M}}^{-1} \in \mathbb{R}^{K\times K}$
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: $\quad\overline{\mathbf{B}}[t-\overline{n}-1] = \lambda\sqrt{\log_+ \frac{t}{\overline{n}}}$ ▷ $\overline{\mathbf{B}} \in \mathbb{R}^{\overline{N}-\overline{n}}$
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$(\overline{\mathbf{b}}, \overline{\mathbf{I}})$ ▷ $\mathbf{b} \in \mathbb{R}^{N-n}$

[3]For all GPU programming aspects, we use CUDA terminology [13].

[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.

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.

2) Our Parallelization Strategy: Our strategy is a mid-point between the two: 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,[5] such that we can translate each such group to a (CUDA/OpenCL) kernel. For example, Line 2 in 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, the full data-parallel hardware-independent specification written in the Futhark language [14] is provided in the appendix. The next two sections discuss two illustrative and performance-critical optimizations. Their impact at kernel and application level is demonstrated in Section IV.[6]

C. Register Tiling of Batch Matrix Multiplication (Like)

Figure 3a shows C-like pseudocode corresponding to computing Line 2 in 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 Ks, i.e., K_1 and 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, but, to our knowledge, it is not supported by any GPU 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., A[j_1,q] is invariant to the loops of indices i and j_2, and Y[i,q] to the loops j_1 and j_2. The result of the transformation is shown in Figure 3b. The

```
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;
            //colective 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. 3: Batched Matrix Multiplication under Variant Y Mask.

essence of it is that the original three (parallel) loops have 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.

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

[6]It is worth mentioning that all optimizations were implemented as general code transformations in Futhark [15]. In fact all code versions were automatically compiled from high-level Futhark specifications, which we believe are accessible to researchers in this field within a gentle learning curve.

[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 (j<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. 4: Batched Matrix Inversion: High-Level Specification

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).

D. Batched Matrix Inversion in Shared Memory

Figure 4 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×(2*K) 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 $3 * K$ fewer accesses to global memory in comparison to a "naive" implementation, which generates a factor of $5-6\times$ speed-up.

E. Implementation Details

The overall implementation of Algorithm 1 is based on Python 3.6 and resorts to the optimized kernels outlined above, which are integrated via PyOpenCL [16]. All GPU code is generated automatically from data-parallel hardware-agnostic Futhark specifications, similar to the one in Figure 11. The incoming data is processed in chunks. For each chunk, the data are copied from host to device prior to invoking the kernels. After the execution of the kernels, the break indices $\mathbf{b}$ are copied back to host. The transfer time for copying the data between host and device is small compared to the kernel execution time and can be completely hided by interleaving the transfer and computing phases.

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 200 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

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

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.

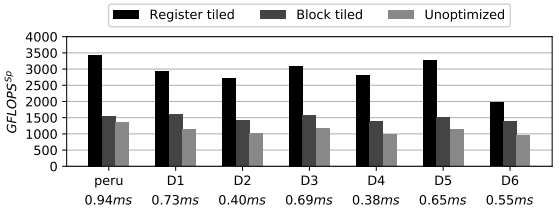


Fig. 5: Performance of Batch-Masked Matrix Multiplication

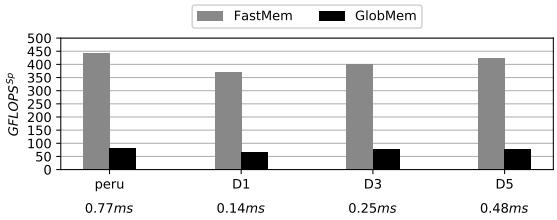


Fig. 6: Performance of Batched Gauss-Jordan Matrix Inversion

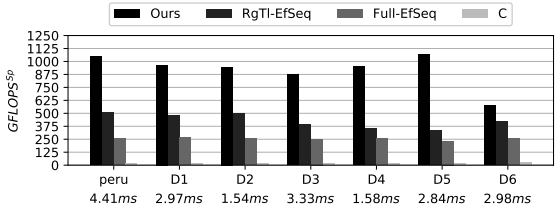


Fig. 7: Performance of one GPU invocation of BFAST.

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.

The performance was analyzed on the datasets shown in Table I; their parameters M, N, n are named as in Algorithm 1, where tt 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, which is common in practice. D6 is a stress case using a rather small n = N/4, and having a high NaN frequency. While datasets D1–D6 are artificial, Peru(Small) is a real-world dataset, which also exhibits a high frequency of invalid values (its description is given below). 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 5 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 $\mathtt{Y}_{[:,:n]}$, as presented in section III-C. 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 [14]), and the third bar refers to no tiling at all (unoptimized)).

The performance of register tiling is relatively stable between $2.7 - 3.5$ 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 $\mathtt{Y}_{[:M,:n]}$. 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 $1.5\times$ speed-up for the other datasets—because the transposition takes roughly the same time as the computation.

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 6 compares, in the case of batched matrix-inversion kernel presented in Section III-D, our strategy of aggressively utilizing fast/shared memory (first bar) to the one that still exploits all parallelism and features with coalesced accesses but utilizes only global memory (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, until one realizes that the ratio 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 7. 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`, where the higher weight of missing values favors efficient sequentialization.

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, 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.

V. Large-Scale Change Detection

A. Experimental Setup and Datasets

We tested our implementation on three datasets of increasing size. All datasets were acquired through the *Google Earth Engine* [17] and consist of Landsat Collection 1 Tier 1 Surface Reflectance derived NDMI images of different sensors (Landsat 5, 7, and 8) [8]. The data was collected only for the pixels that were considered to be forest in 2010, using the global tree cover with a threshold of 30 [18], see Figure 8.

- *`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 of palm oil plantations. 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 the entire province of Padre Abad in the central Amazon rainforest of Peru. 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 from $N = 6873$ dates between 01/01/2000 and 31/12/2018.

In addition to the performance analysis reported above, we evaluated the practical runtime of the different phases. For these runtime experiments, we resorted to 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).

B. Practical Runtime Analysis

The practical runtimes of our implementation (based on the **Ours** kernels) on the three datasets is shown in Figure 9. We measured the preprocessing time (host), the transfer time (between host and GPU), and the kernel execution time (GPU). The single images of both `Peru(Large)` and `Africa` were too big to fit into GPU memory and were, hence, split into 50 chunks. The overhead for this chunking (host) is shown as well. The runtimes depict averages over ten runs for both `Peru(Small)` and `Peru(Large)`. For `Africa`, the average runtime over 50 random images is reported. 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 analyzed in the previous section). It is worth noting that loading the images from disk to host has now become the bottleneck (e.g., about 25s for `Peru(Large)`). However, from a practical perspective, this is acceptable since for each image that is loaded into memory, many BFAST runs are usually conducted (e.g., for different values for k or for different monitoring periods, ...), which compensates the overhead for loading the data.

C. Change Detection for Massive Data

Finally, we conducted large-scale experiments for all datasets (the end of the history period was set to 2010 and different years were considered for the monitoring period). In addition to the break indices (and the associated time), BFAST 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 are shown in Figure 8 for `Peru(Small)` and `Peru(Large)` and in Figure 10 for `Africa`, respectively. The figures show where and when a break with negative magnitude occurred, which can be interpreted as vegetation decrease after 2010. Using our new implementation, it takes about three days to process a single monitoring period (e.g., 2010–2011), and about four weeks to process the whole scenario using a single GPU. 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.[9]

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.

[9] It takes about six hours to process `Peru(Large)` using the `R` implementation and a powerful 36 core machine. The GPU implementation needs about 60 seconds to process all the four monitoring phases shown in Figure 8, which yields a speed-up of about 360 compared to the `R` implementation. Thus, instead of four weeks, it would take about thirty years to obtain the results.

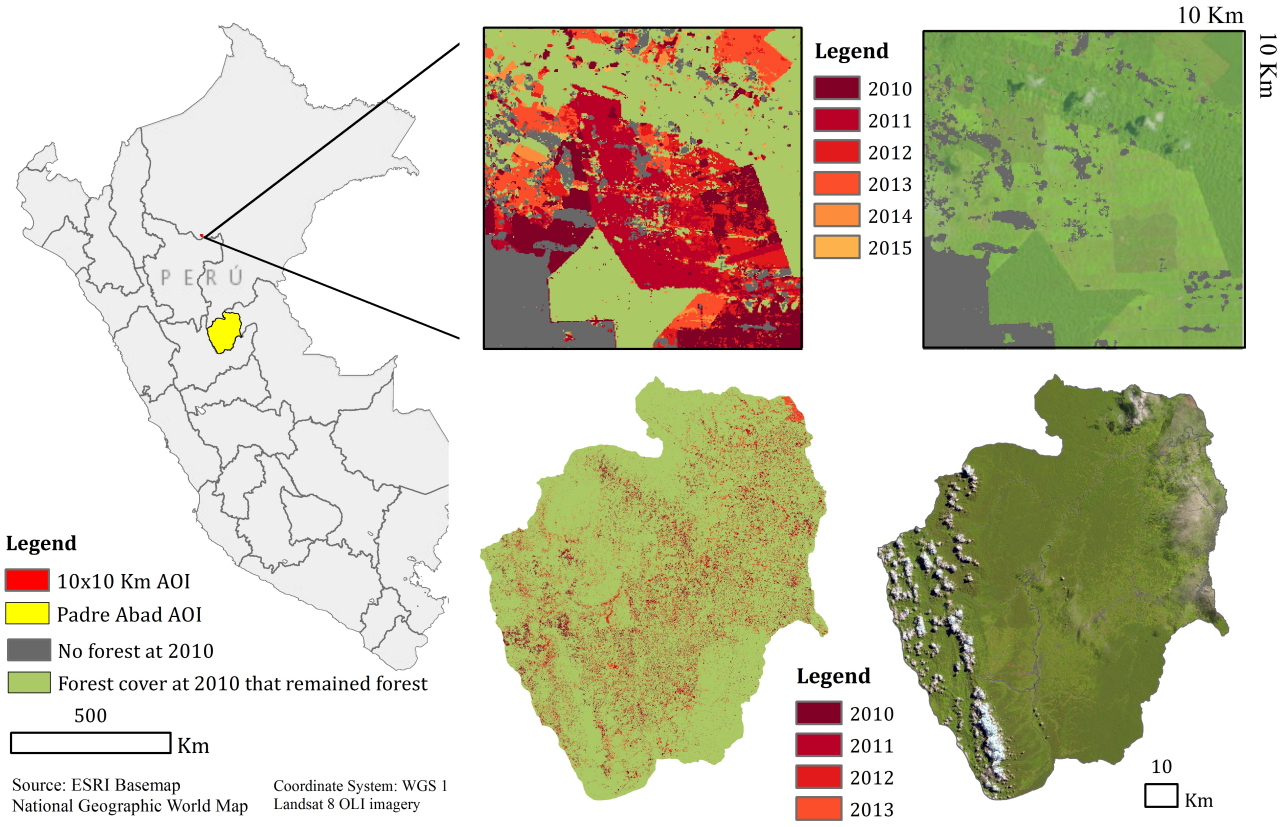
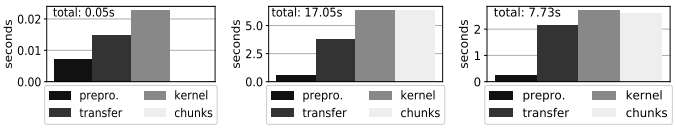


Fig. 8: The `Peru(Small)` and `Peru(Large)` datasets and the results obtained via BFAST. The history period was from 2000 till 2010, whereas different ranges were considered as monitoring periods (2010–2011, 2011–2012, ...).



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

Fig. 9: Practical runtimes of our implementation.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] T. Henriksen, F. Thore, 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.

[16] 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.

[17] 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.

[18] 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

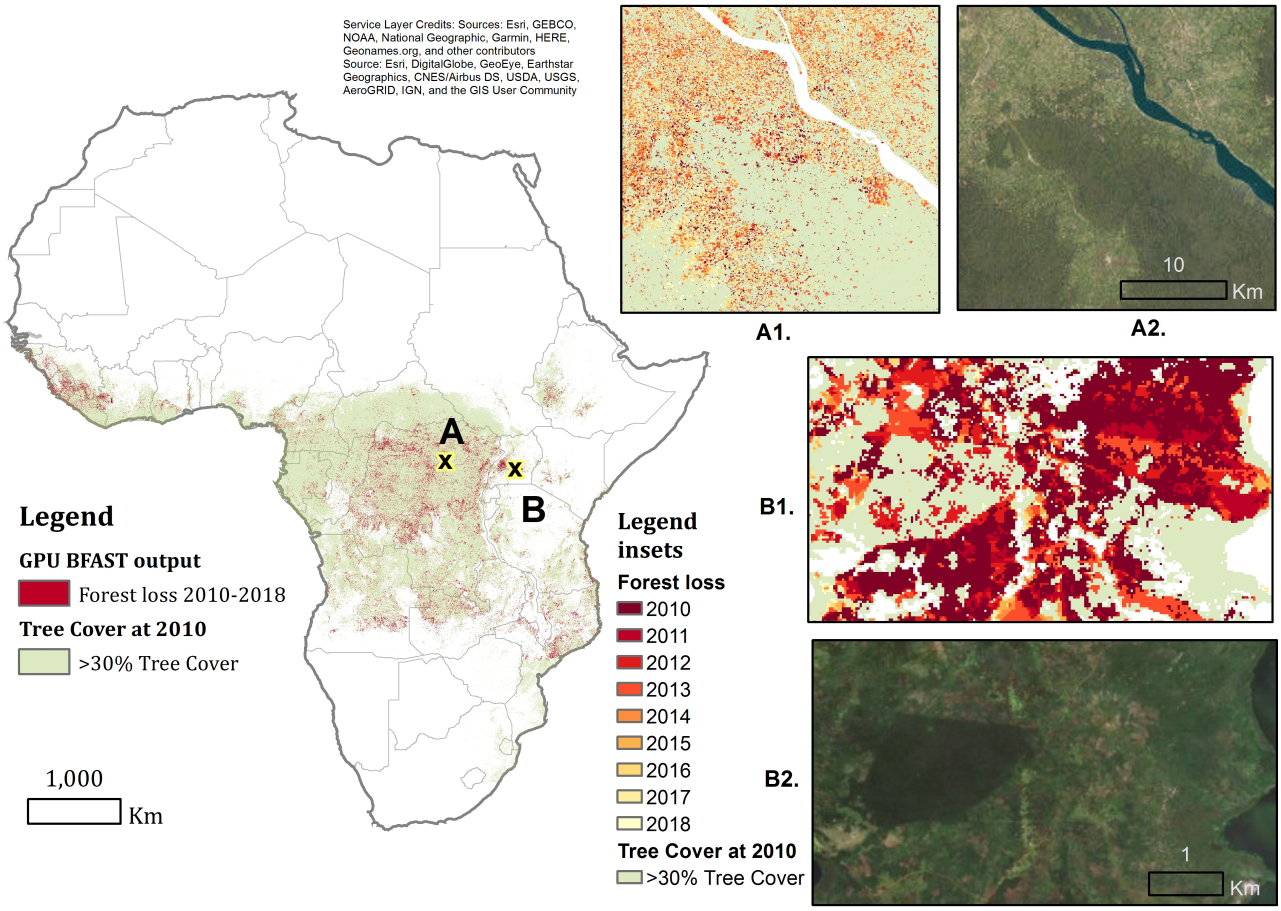


Fig. 10: Large-scale analysis covering the entire continental tropical Africa.

the type of a two-dimensional array of floats with `N` rows and `M` columns. We use `[a1,...,an]` 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. The semantics of several parallel operators is presented below:

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

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 [12] 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 11, 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 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-C and III-D. 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 11, 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.[10]

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 $\mathtt{m}^{fst}$ 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.

[10]The implementation of `filterNaNsWKeys`, shown in Figure 11, 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; β̄_0, β̄:[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 β̄_0  = mvMulFilt X_[:,:n] y_[:n]       -- ker 4
       let β̄    = mvMul X^sqr^-1 β̄_0            -- ker 5
       -- ŷ, r̄, Ī : [N]f32
       let ŷ    = mvMul X^T β̄                   -- 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. 11: Nested Data-Parallel Structure of BFAST. Functions `r32` and `t32` implement conversions between real and integral values. For readability purposes, function `isnan` returns, according to context, 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-C and III-D.