2023 IEEE International Conference on Big Data (BigData) | 979-8-3503-2445-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/BigData59044.2023.10386208

Seasonal-Trend Time Series Decomposition on Graphics Processing Units

Dmitry Serykh, Stefan Oehmcke*, Cosmin Oancea†
Dept. of Computer Science
University of Copenhagen
{*stefan.oehmcke, †cosmin.oancea}@di.ku.dk

Dainius Masiliūnas‡, Jan Verbesselt§
Dept. of Information Systems
Wageningen University
{‡dainius.masiliunas, §jan.verbesselt}@wur.nl

Yan Cheng¶, Stéphanie Horion‖
Dept. of Geosciences and Natural Resource Management
University of Copenhagen
{¶ych, ‖smh}@ign.ku.dk

Fabian Gieseke
Dept. of Information Systems
University of Münster
fabian.gieseke@uni-muenster.de

Nikolaj Hinnerskov
Dept. of Computer Science
University of Copenhagen
nihi@di.ku.dk

***Abstract*—In many domains, large amounts of time series data are being collected and analyzed in a semi-automatic manner. A prominent approach is the seasonal and trend decomposition using locally estimated scatterplot smoothing (STL) technique, which has been applied extensively in the past. However, STL quickly becomes computationally very expensive when applied to large data sets. In this work, we propose the first parallel implementation for the STL decomposition approach, which is tailored to the specific needs of graphics processing units (GPU). Our experimental evaluation on two global-scale case studies in temperature and vegetation trend analysis exhibits at least three-to-four orders of magnitude speed-up, demonstrating the effectiveness of the overall approach and the immense potential of the implementation in spatio-temporal data analyses. The source code is publicly available at https://github.com/diku-dk/hastl. An artifact that allows the experimental results to be reproduced is available at https://sid.erda.dk/sharelink/hOUrqJJFfA.**

***Index Terms*—Time Series Data, Remote Sensing, Climate Change, Trend Analysis, Parallel Implementation**

I. Introduction

Many domains witness an explosion in the amount of temporal-spatial data. For instance, in climate and environmental research, new satellites and other observational systems are producing terabytes or even petabytes of data every day [6], [26], [31], [34], [41]. This often results in large time series data sets that allow us to monitor and detect changes on Earth, such as heatwaves, major flooding, drought, or deforestation events [53]. Time series data are also abundant in many other domains, such as in astronomy [18], industry, and commerce (e.g., stock market analysis) [33]. Today, these data are often analyzed in a semi-automatic fashion via machine learning techniques. For instance, modern deep neural networks [11], [23], [30], [52] can be applied to detect rare events or changes in time series data. However, large amounts of labeled time series data are typically needed to train corresponding models in a supervised manner to achieve satisfying results. In contrast, unsupervised methods do not require labelled data and are, hence, often used at early stages of the analyses [19].

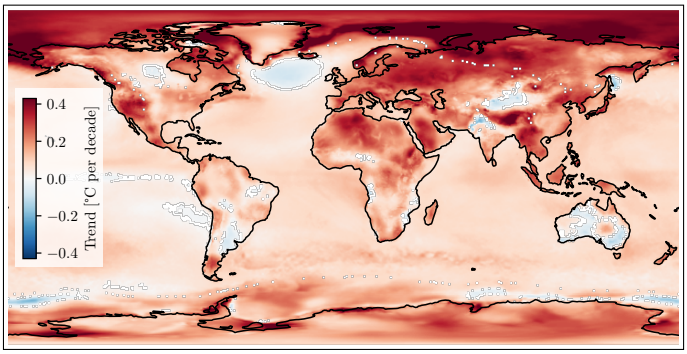


Fig. 1: The figure demonstrates the use of STL to extract trends in time series data at large scale. Depicted are global trends in midday temperature time series data derived from the Copernicus ERA5 land data set. For such global analyses, millions of pixels—each corresponding to some time series—often have to be analyzed, which presents a substantial computational challenge. STL was applied with smoothing factors $q_s = 365$ and $q_t = 7301$ corresponding to one- and 20-year windows, respectively. A t-test with a significance level of 5% was carried out for each pixel using `statsmodels.OLS` and trends found to be insignificant were masked out.

The seasonal and trend decomposition using locally estimated scatterplot smoothing (STL) method [8], [17] is such an unsupervised approach, which can be used to analyze time series data. In particular, it decomposes a time series into a seasonal, a trend, and a remainder component. The STL approach is known to be a versatile, robust, and highly adjustable time series decomposition algorithm that can be employed as a standalone algorithm or as an algorithmic building block for machine learning pipelines [23], forecasting

This work has been supported by the Independent Research Fund Denmark (DFF) under the grant: High-performance Architectures and Monitoring Changes in Big Satellite Data via Massively Parallel AI, and by the UCPH Data+ grant: High-Performance Land Change Assessment.

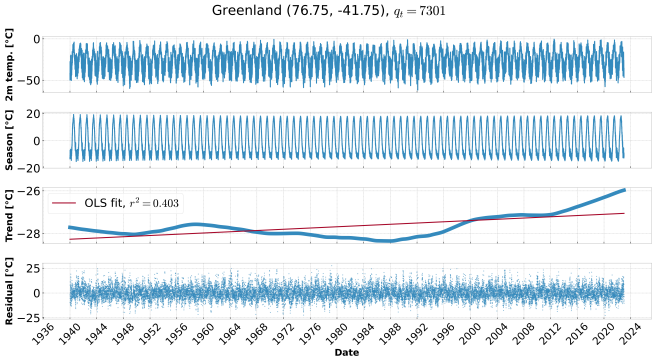


Fig. 2: The top row shows a time series of midday temperatures measured at 2 m above ground-level for a single location in Greenland. The STL approach outputs a decomposition of such time series into a seasonal $\mathbf{s}$, a trend $\mathbf{t}$, and a remainder $\mathbf{r}$ component. The trend component is summarized by a least-squares fit of a first degree polynomial.

scenarios [46], Earth science [42], scientific visualization [2], or remote sensing [3], [13], [47]. It is also subject of ongoing research, including the development of "robust" versions [50] or the acceleration of sequential implementations [51].

For many application scenarios, hundreds of millions of time series have to be analyzed. Figures 1 and 2 illustrate one of the most common outputs in climate research; an assessment of long-term changes in temperature over hundreds of millions of locations/pixels at the global scale. A major drawback of the STL approach (and similar schemes) is that its application to large amounts of data can quickly become very intensive from a computational perspective, which usually limits the analyses to small-scale scenarios. Existing STL implementations are also not tailored to the specific requirements of modern parallel compute devices, such as graphics processing units (GPUs), especially for time series with many elements.

In this work, we present a memory- and runtime-efficient parallel implementation of the STL decomposition algorithm that is suited for modern GPU hardware. In particular, we show how to translate the mathematical operations needed to fit STL models into a data-parallel implementation that (i) can, in principle, exploit all levels of available parallelism and that (ii) is also capable to adapt to hardware and dataset characteristics by efficiently sequentializing some of the excess parallelism to better exploit locality of reference.[1] The implementation—which enables efficient execution of batches of time series of any length on Nvidia and AMD GPUs—is provided to the user in the form of an easy-to-use Python package[2].

We demonstrate the effectiveness of our approach in the context of two large-scale case studies in the geospatial domain and show that our GPU implementations are three-to-four orders of magnitude faster than existing (CPU) solutions. This renders, for the first time, such large-scale analyses possible using standard (cheap) commodity hardware. In addition, we report a performance analysis that shows that our implementation achieves on average 89.6% and as high as 148% of the peak memory bandwidth of an Nvidia A100 GPU. (Figures higher than 100% are possible due to the exploitation of temporal locality, i.e., reuse from L2 cache.)

[1]We achieve this by building multiple semantically-equivalent but differently optimized code versions that are discriminated at runtime based on a one-time autotuning process.

[2]We intend to further integrate it into the popular TIMESAT framework [13].

II. Related Work

The STL decomposition approach is an essential time series analysis tool that has been extensively used [1], [4], [5], [43] since its development in 1990 [8]. For example, STL has been employed as a preprocessing step for machine learning methods [45] and as a building block of other time series processing techniques, including landscape change detection methods in remote sensing [47] or bagging for exponential time series smoothing in statistics, forecasting, and machine learning [3]. As illustrated by the remote sensing case study in Section V, these applications demand the efficient processing of increasingly large, planet-scale datasets. Our work aims to meet this demand.

Time series decomposition is still an active research topic. RobustSTL, an adaption of classic STL, focuses on developing a new algorithm for time series decomposition that can better deal with long seasonality period and missing values [50]. However, as observed by Wen *et al.* [51], RobustSTL suffers from high computational complexity, limiting its application in practice. This motivates the authors to propose Fast RobustSTL, which improves on the runtime bounds; unlike in our work, however, GPU parallelization is not discussed, and the evaluation of the new method is restricted to small datasets with only a few thousands of observations. Similar remarks can be made about decomposition approaches based on new methodologies, such as neural networks [16]. In spite of the algorithmic improvements in these methodologies, according to Wen *et al.* [51], STL as studied in this paper remains the "most classical and widely used decomposition method", justifying our focus on this algorithm. Moreover, our work shows that different data sets and hardware have a significant impact on the efficiency of parallelization, necessitating different code versions of STL in order to achieve good performance. This observation made in our work illustrates that parallelization of seasonal-trend decomposition approaches is a non-trivial endeavor.

III. Background

We start by providing a short description of the original STL approach. We refer the reader to Cleveland *et al.* [8] for a more in-depth description.

A. Sequential Algorithm

STL is an additive seasonal trend decomposition method that splits up a given time series $\mathbf{y} \in \mathbb{R}^N$ into a seasonal $\mathbf{s} \in \mathbb{R}^N$, a trend $\mathbf{t} \in \mathbb{R}^N$, and a remainder $\mathbf{r} \in \mathbb{R}^N$

Algorithm 1 STL

1: **Input:** Time series $\mathbf{y} \in \mathbb{R}^N$ and parameters n_p, n_{inner}, n_{outer}, q_s, q_t, q_l, d_s, d_t, d_l, n_{jump_s}, n_{jump_t}, and $n_{\text{jump}_l} \in \mathbb{Z}^+$
2: **Output:** Seasonal trend $\mathbf{s} \in \mathbb{R}^N$, trend component $\mathbf{t} \in \mathbb{R}^N$, and remainder component $\mathbf{r} \in \mathbb{R}^N$
3: Initialize $\mathbf{s}, \mathbf{t}$ to $\mathbf{0}$, $\mathbf{w} \in \mathbb{R}^N$ to $\mathbf{1}$ and $\mathbf{x}$ to $[0, 1, ..., N-1]$
4: **for** $i_{\text{outer}} = 1$ **to** n_{outer} **do**
5: **for** $i_{\text{inner}} = 1$ **to** n_{inner} **do**
6: $\mathbf{z} = \mathbf{y} - \mathbf{t}$ // $\mathbf{z} \in \mathbb{R}^N$
7: $N_c = N + 2\,n_p$
8: Initialize $\mathbf{c} \in \mathbb{R}^{N_c}$ to $\mathbf{0}$
9: **for all** $i = 0$ **to** $n_p - 1$ **do**
10: $\mathbf{z}_c$, $\mathbf{w}_c$, $\mathbf{x}_c$ = EXTRACTCSS($\mathbf{z}, \mathbf{w}, \mathbf{x}, i$)
11: $\overline{\mathbf{z}}_c$, $\overline{\mathbf{w}}_c$, $\overline{\mathbf{x}}_c$ = FILTERNANS($\mathbf{z}_c, \mathbf{w}_c, \mathbf{x}_c$)
12: $\mathbf{c}[i : N_c : n_p]$ = LOESS($\overline{\mathbf{x}}_c, \overline{\mathbf{z}}_c, \overline{\mathbf{w}}_c, q_s, d_s, n_{\text{jump}_s}$)
13: $\mathbf{c}_l$ = MOVINGAVERAGES($\mathbf{c}$) // $\mathbf{c}_l \in \mathbb{R}^N$
14: $\mathbf{l}$ = LOESS($\mathbf{x}$, $\mathbf{c}_l$, $\mathbf{w}$, q_l, d_l, n_{jump_l}) // $\mathbf{l} \in \mathbb{R}^N$
15: $\mathbf{s} = \mathbf{c}[n_p - 1 : N + n_p] - \mathbf{l}$
16: $\mathbf{u} = \mathbf{y} - \mathbf{s}$ // $\mathbf{u} \in \mathbb{R}^N$
17: $\overline{\mathbf{u}}$, $\overline{\mathbf{w}}$, $\overline{\mathbf{x}}$ = FILTERNANS($\mathbf{u}, \mathbf{w}, \mathbf{x}$) // $\overline{\mathbf{x}}, \overline{\mathbf{u}}, \overline{\mathbf{w}} \in \mathbb{R}^n$
18: $\mathbf{t}$ = LOESS($\overline{\mathbf{x}}$, $\overline{\mathbf{u}}$, $\overline{\mathbf{w}}$, q_t, d_t, n_{jump_t})
19: $\mathbf{r} = \mathbf{y} - \mathbf{t} - \mathbf{s}$
20: **if** $i_{\text{outer}} < n_{\text{outer}}$ **then**
21: $\mathbf{g} = |\mathbf{r}| \,/\, (6\,\text{median}(|\mathbf{r}|))$ // $\mathbf{g} \in \mathbb{R}^N$
22: **for all** $i = 0$ **to** $N - 1$ **do**
23: $\mathbf{w}[i] = (1 - g[i]^2)^2$ **if** $0 \le \mathbf{g}[i] < 1$ **else** 0

Algorithm 2 LOESS

1: **Input**: Timesteps, timeseries and weights $\mathbf{x}, \mathbf{y}, \mathbf{w} \in \mathbb{R}^n$, and parameters $q, d, n_{\text{jump}}, N \in \mathbb{N}$
2: **Output**: Smoothened timeseries $y_{out} \in \mathbb{R}^N$
3: Initialize $\mathbf{y}_{\text{results}}, \mathbf{y}_{\text{slopes}} \in \mathbb{R}^n$ with 0
4: $n_m = \lceil N / n_{\text{jump}} \rceil$
5: **for all** $j = 0$ **to** $n_m - 1$ **do**
6: $\hat{x}_j = j \cdot n_{\text{jump}}$
7: $\hat{q} = \min(q,\, n)$
8: $l_{\text{idx}}, r_{\text{idx}}$= NNS($\hat{x}_j$, $\hat{q}$, $\mathbf{x}$) // Computes the $\hat{q}$ nearest neighbors
9: $\lambda_q = \max(q/n, 1) \cdot \max(|l_{\text{idx}} - \hat{x}_j|, |r_{\text{idx}} - \hat{x}_j|)$
10: $\overline{\mathbf{x}}$, $\overline{\mathbf{y}}$, $\overline{\mathbf{w}}$ = QSLICE($\mathbf{x}, \mathbf{y}, \mathbf{w}, l_{\text{idx}}, \hat{q}$)
11: $\tilde{\mathbf{x}} = \overline{\mathbf{x}} - \hat{x}_j$ // $\tilde{\mathbf{x}} \in \mathbb{R}^q$
12: $\mathbf{v} = (1 - (|\tilde{\mathbf{x}}|/\lambda_q)^3)^3$ // $\tilde{\mathbf{v}} \in \mathbb{R}^q$
13: $\hat{\mathbf{w}} = \overline{\mathbf{w}} * \mathbf{v}$ // $\hat{\mathbf{w}} \in \mathbb{R}^q$
14: $\mathbf{y}_{\text{results}}[j], \mathbf{y}_{\text{slopes}}[j]$ = WEIGHTOLS($\tilde{\mathbf{x}}$, $\overline{\mathbf{y}}$, $\hat{\mathbf{w}}, d$)
15: **if** $n_{\text{jump}} > 1$ **then**
16: $\mathbf{y}_{\text{out}}$ = INTERPOLATE($\mathbf{y}_{\text{results}}, \mathbf{y}_{\text{slopes}}, n_{\text{jump}}, N$)
17: **else**
18: $\mathbf{y}_{\text{out}} = \mathbf{y}_{\text{results}}$

component, see again Figure 2. The overall decomposition algorithm is based on two nested loops, see Alg. 1: The inner loop (Lines 5–18) updates both the seasonal $\mathbf{s}$ and the trend $\mathbf{t}$ component, while the outer loop (Lines 4–23) updates a set of robustness weights based on the current estimate of the remainder component $\mathbf{r} = \mathbf{y} - \mathbf{t} - \mathbf{s}$. Loops that are already fully parallelizable are annotated by **for all**.

1) Algorithmic Workflow: The outer loop (Lines 4–23) simply runs the inner loop and updates the remainder term $\mathbf{r} = \mathbf{y} - \mathbf{t} - \mathbf{s}$. Robustness is introduced into STL by giving each data point a robustness weight w_i (Lines 20–23). These weights express the reliability of an observation and are calculated from the current estimate of the remainder component $\mathbf{r}$. If $|r_i|$ exceeds the value of $h = 6 \cdot \text{median}(|\mathbf{r}|)$, its robustness weight is set to 0, indicating that the observation is not reliable. Otherwise, a robustness weight $w_i \in (0, 1)$ is computed using $w_i = B(|r_i|/h)$, where B is the *bisquare weight function* $B(u) = (1 - u^2)^2$. The resulting vector $\mathbf{w} \in \mathbb{R}^N$ is used for locally estimated scatterplot smoothing (LOESS) in the inner loop, as explained next.

The inner loop (Lines 5–18) computes a new estimate for the trend and seasonal components in several steps:

1) First, the difference $\mathbf{z} = \mathbf{y} - \mathbf{t}$ between the time series and the trend component is considered (if y_i is missing, the corresponding z_i is also missing), i.e., the time series is "detrended" given the current trend estimate.
2) Next, cycle subseries are computed in Lines 9–12. A cycle subseries is a time series obtained by taking the values corresponding to observations at each position of a seasonal cycle.[3] The procedure EXTRACTCSS extracts all n_p cycle subseries from the detrended time series $\mathbf{z}$, together with the corresponding robustness weights $\mathbf{w}$, and time steps $\mathbf{x}$. Then, missing values are filtered out by the procedure FILTERNANS. Afterwards, a smoothing procedure LOESS (sketched in Section III-A2) is applied to each cycle subseries. This smoother considers the subseries with two extra values: one just prior to the first observation and one just after the last. The smoothed time series are then combined into a vector $\mathbf{c}$ that contains no missing values.
3) Afterwards, a low-pass filter is applied to $\mathbf{c}$ via the function MOVINGAVERAGES—followed by LOESS (see Section III-A2). The values of resulting vector $\mathbf{l}$ are then subtracted from $\mathbf{c}$, yielding the current estimate for $\mathbf{s}$.
4) Finally, the trend component $\mathbf{t}$ is computed by subtracting the seasonality component from the time series (Line 16), by removing missing values from $\mathbf{u}$, $\mathbf{x}$ and $\mathbf{w}$ (Line 17), and by applying LOESS (Line 18).

One of the key algorithmic building blocks of the STL algorithm is the LOESS procedure [9], which is described next.

2) LOESS: The LOESS procedure removes an adjustable amount of high-frequency information from a given time series, thus making its curvature less pronounced. Figure 3 displays the application of LOESS to harmonic data: as the smoothing factor q increases, more smoothing is applied. Let the input time series be represented as a vector of measurements $\mathbf{y} = (y_1, \ldots, y_N) \in \mathbb{R}^N$ taken at times $\mathbf{x} = (x_1, \ldots, x_N) \in \mathbb{R}^N$. Let $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_n) \in \mathbb{R}^n$ constitute the points at which the smoothing is applied. Note that $\hat{\mathbf{x}}$ is not necessarily a subset of $\mathbf{x}$. Then, for each $\hat{x}_j \in \hat{\mathbf{x}}$, the value of a LOESS fit is computed in three steps:

1) Given a positive, odd integer q, find the q nearest neighbors of $\hat{x}_j \in \mathbb{R}$ among the values $x_1, \ldots, x_N$. Specifically, procedure NNS (Line 8 in Alg. 2) computes the start and end indices $(l_{\text{idx}}, r_{\text{idx}})$ that slice $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{w}$ to this nearest-neighbors range using QSLICE (Line 10

[3]For example, in a monthly data set with yearly periodicity, the first cycle subseries would contain the January values, the second the February values, and so on.

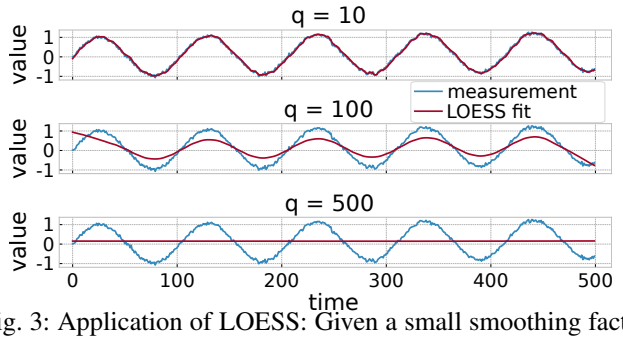


Fig. 3: Application of LOESS: Given a small smoothing factor q, the LOESS fit overfits to the data (little smoothing). The larger q, the smoother gets the fit. Hence, q can be seen as regularization parameter for the LOESS fit.

in Alg. 2).

2) Assign each of the chosen neighbors x_i a neighborhood weight v_i (Line 12 in Alg. 2) such that points adjacent to $\hat{x}_j$ receive weights close to one, and the weights decrease as the distance from $\hat{x}_j$ increases. Specifically, $v_i = \mathcal{V}(x_i, \hat{x}_j) = W\left(|x_i - \hat{x}_j|/\lambda_q(\hat{x}_j)\right)$, s.t. $(0 < v_i \leq 1)$, where W is the tricube weight function $W(u) = (1 - u^3)^3$ and $\lambda_q(\hat{x}_j)$ is the distance between $\hat{x}_j$ and its q'th nearest neighbor among the values of $\mathbf{x}$. However, for $q{>}N$, $\lambda_q(\hat{x}_j) = \frac{q}{n} \cdot \lambda_N(\hat{x}_j)$, where $\lambda_N(\hat{x}_j)$ is the distance from $\hat{x}_j$ to the furthest point in $\mathbf{x}$.
3) Fit a polynomial of degree $d \in \{0, 1, 2\}$ to the q nearest neighbors in $\mathbf{x}$ and the corresponding y_i values with weights $v_i \cdot w_i$ using ordinary least squares (WEIGHTOLS, Line 14 in Alg. 2).

The parameter n_{jump} sets a stride for iterating over $x_1, \ldots, x_N$. Skipped values are interpolated using INTERPOLATE (Line 16 in Alg. 2). Further implementation details are provided in Section IV-C.

IV. PARALLEL (GPU) IMPLEMENTATION OF STL

This section presents a *batched* parallel implementation of STL — named STLBATCHED and aimed at execution on GPUs — that (semantically) applies STL (Alg. 1) in parallel to m time series of size N. This is particularly relevant in the remote-sensing field, where common datasets require to analyze millions-to-billions of time series. However, the (conceptually) straightforward approach of utilizing only the batch parallelism (i.e., one thread computes one time series) is misguided for two main reasons: First, the amount of GPU memory and the time-series length restrict the number of time series (hence of threads) that can be processed in parallel at a time, possibly leading to severe under-utilization of the GPU.[4] Second, even when enough active threads exist, this approach features poor spatial[5] and temporal locality (due to high-reuse distance).

A. Parallelization Strategy

It follows that an efficient implementation must exploit both the trivial parallelism of the batch[6] *and* of the inner levels of the STL algorithm, but this is not possible in the current form because they are separated by two sequential loops.[7] To this effect, we interchange the parallel loop of the batch inside the nested sequential loops and then we distribute them across the inner-parallel constructs of STL, thus forming perfectly-nested parallel loops that can be mapped to the GPU hardware. The resulting computational kernels can exploit (if needed) all levels of application parallelism.

In terms of parallel structure, STLBATCHED is a nested composition of `map`, `reduce`, and `scan` (prefix sum) parallel constructs. For instance, the step of smoothing the cycle subseries (of main LOESS) reaches four levels of nested parallelism: the map over the time series, the map over the cycle subseries, the map over the observations, and the map-reduce computations over the q nearest neighbors.

The main challenge in devising an efficient implementation for STLBATCHED is that there is no "one size fits all" optimization recipe. For example, Section IV-C4 shows that a code version that is optimal for a class of datasets may be $20\times$ slower on others[8], and performance is not portable across different GPUs.

We address this issue by maintaining multiple semantically equivalent but differently optimized code versions for each computational kernel and discriminating between them at runtime, based on a one-time per-hardware autotuning process. We use Futhark [20], [37] as implementation language and partially rely on its support for multi-version compilation [21] and autotuning [38].

The next sections comment on the parallel specification and optimization opportunities of the individual STL components: LOESS is treated in Section IV-C and the remaining kernels in Section IV-B.

B. Filter, Moving Averages & Median Kernels

1) NaN Filtering (filter): We separate the filtering of NaN values (Lines 11 and 17 in Alg. 1) into two parts: the calculation of indexes of the non-NaN values and a (cheaper) gather/padding operation that packs the non-NaN values consecutively into a new array of the same size and pads the

[4]Nvidia A100 GPU requires 100K of active threads for full utilization, but time-series lengths in the 10K-to-million range would only permit thousands-to-tens of threads.

[5]LOESS would exhibit prohibitively-expensive uncoalesced accesses, because their indices depend on the nearest neighbor result, which is statically unpredictable.

[6]Batches of time series can also be trivially distributed across multiple GPUs.

[7]E.g., the parallel batch loop that should encompass the code of Alg. 1 is separated from the parallel (**for all**) loop at line 9 by two sequential (**for**) loops at lines 4 and 5.

[8]Interestingly, for the same computational kernel, locality of reference is better exploited (1) on some datasets when following the common wisdom of efficiently sequentializing inner levels of (excess) parallelism, and (2) on other datasets when all levels of parallelism are utilized (similar observations are made in [15], [39]).

```
[n]real single_ma(long n_p, long n, []real x)=
  s = sum(x[:n_p])          # segmented reduce
  real arr[n]
  forall i = 0 to n-1 do    # segmented map
    arr[i] = ((i==0) ? s : x[i+n_p-1]-x[i-1]) / n_p
  return scan(+, 0, arr)    # segmented prefix sum

[m][n-2*n_p]real mov_avgs([m][n]real cs, long n_p)=
  real rs[m][n-2*n_p]
  forall i = 0 to m-1 do    # outer map
    x    = single_ma(n_p, n-n_p+1  , cs[i])
    y    = single_ma(n_p, n-2*n_p+2, x)
    rs[i]= single_ma(  3, n-2*n_p  , y)
  return rs
```

Fig. 4: Moving-Averages Parallel Pseudocode. `[m][n]real` denotes a $m \times n$ matrix of reals.

	A_1	A_2	A_3	B_1	B_2	B_3	C_1	C_2	C_3
loess_main	58.47	68.85	62.01	53.32	61.07	52.76	67.76	73.51	71.83
nns	20.54	14.93	6.28	25.24	18.84	9.17	18.12	14.82	7.82
interp	10.83	6.66	10.75	12.01	11.23	10.43	7.81	6.31	8.69
filter	6.38	6.01	5.20	5.59	5.25	4.52	3.96	3.23	3.89
ma	3.78	3.55	3.76	3.84	3.61	3.79	2.35	2.13	2.53
median	0.00	0.00	12.00	0.00	0.00	19.32	0.00	0.00	5.24

TABLE I: Estimate of the percentage of runtime for each of the main computational kernels. The estimates are obtained using sequential C execution. The reported kernels cover over 90% of the runtime; the percentages are relative to the sum of the shown kernels. The datasets are detailed in Section V.

remaining values with zeroes. The former part is invariant to the two sequential loops and is only computed twice: once for the whole input and once for the cycle subseries. The padding/-gathering is a simple parallel map that is executed on every invocation of FILTERNANS. The cycle subseries extraction (EXTRACTCSS at Line 10 in Alg. 1) is essentially a change-of-layout operation (transposition) that is not manifested in memory and is fused with the gather operation.

2) Moving Averages (ma): Figure 4 shows the pseudocode for the moving averages function (called at Line 13 in Alg. 1). The helper function `single_ma` consists of a reduce (`sum` operation), followed by a map (`forall`) whose result is piped into a prefix-sum (`scan`) operation.[9] The implementation of moving averages consists of chaining `single_ma` three times on different sizes, for each time series of the batch. The runtime of this stage is dominated by the segmented map-scan computations. We have added support in the Futhark compiler [7] for a scan implementation inspired by the single-pass-scan [36], but extended it to also support regular-segmented scans and scan-map fusion. CUB [35] supports neither extension directly; our implementation for prefix sum alone is competitive with CUB, reaching $\sim 80\%$ of `memcpy`'s speed.

3) Selection algorithm (median): The runtime of the robustness weights computation is dominated by the median computation, called at Line 21 in Alg. 1. We compute medians using a flat-parallel implementation of a batched version of the quickselect algorithm [24], which offers comparable performance with CUB's radix sort [35], and even outperforms it on short time series.

C. Implementation of LOESS

We base our LOESS implementation on *stlplus* [17], a popular STL implementation, but we introduce a number of algorithmic changes that improve performance and reduce the memory footprint.

1) Jump Values: Alg. 2 shows the pseudocode for the LOESS procedure. The application of LOESS can be accelerated using the optimization suggested by Cleveland *et al.* [8]. Instead of computing LOESS for each point in the time series, one can skip a constant number of observations. Parameter n_{jump} dictates how many observations are skipped, e.g., if $n_{\text{jump}} = 2$, LOESS is applied to $x_1, x_3, x_5, \ldots$ and the intermediate values are interpolated. This modification reduces the number of fitted polynomials from N to $n_m = \lceil N/n_{\text{jump}} \rceil$, speeding up execution considerably.

2) Nearest neighbor search (nns): The first step of LOESS is to find the q nearest neighbors for each $\hat{x}_j \in \hat{\mathbf{x}}$. Since $\mathbf{x}$ is sorted, the neighbors must be consecutive in the filtered time series. Function NNS (Line 8 in Alg. 2) returns the leftmost (l_{idx}) and rightmost (r_{idx}) indices of the interval on the x-axis that contains the q nearest neighbors of $\hat{x}_j$. Then, λ_q is easily computed, since either l_{idx} or r_{idx} is the q'th nearest neighbor of $\hat{x}_j$. We start by using binary search to find the closest point to $\hat{x}_j$, named x_c in $O(\log N)$ steps. Then, we find l_{idx} and r_{idx} in q steps, by looking left and right and extending the nearest neighbors interval one point at a time.

We thus find the nearest neighbors interval in $O(q+\log N)$ steps and $O(1)$ extra space, while *stlplus* [17] uses kd-trees and has $O(q)$ space complexity. This modification is crucial for large scale applications, since q can be as large as the time-series length (N).

The nearest neighbor search and the calculation of λ_q is invariant to $\mathbf{y}$. Hence, we compute l_{idx} and λ_q for each $\hat{x}_j \in \hat{\mathbf{x}}$ and for each time series *prior to the outer loop* of STL, by means of a parallel loop (map) of total size $m \cdot n_m$ which has depth $logN + q$ (i.e., the binary search and the interval expansion are sequential). Since $\hat{x}_j \in \hat{\mathbf{x}}$ are sorted, the threads (accesses) benefit from temporal-locality reuse.

3) Interpolation (interp): Function INTERPOLATE (Line 16 in Alg. 2) is based on the *stlplus* [17] implementation and uses $O(1)$ extra memory. This method uses the LOESS-smoothed values and first derivatives, returned by the WEIGHTEDOLS function, and implements the cubic Hermite interpolation [14]. The output is a continuous, differentiable curve of size N. The implementation consists of a parallel map executing a series of floating point operations.

4) Main LOESS Procedure (loess_main): This computational kernel corresponds to lines $10-14$ in Alg. 2 and consists of (i) the extraction of the values corresponding to the q nearest neighbors from $\mathbf{x}, \mathbf{y}$, and $\mathbf{w}$, (ii) the calculation

[9] `scan(`$\oplus$`,`$e_\oplus$`,[`$x_1,\ldots,x_n$`])` $= [x_1, x_1 \oplus x_2, \ldots, x_1 \oplus \ldots \oplus x_n]$

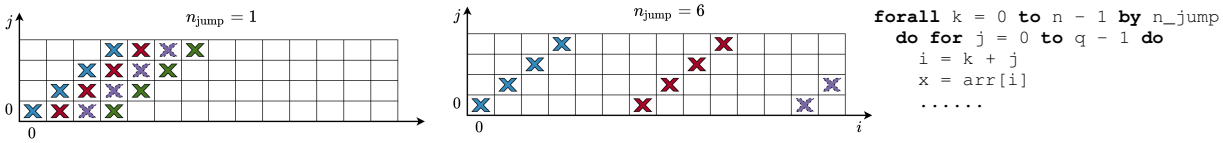


Fig. 5: Demonstration of the access patterns of the simplified QSLICE function in the outer-parallel version of the main LOESS procedure. Here, j is the index of the inner sequential loop, k is the index of the outer parallel loop, and $i = k + j$ is the index of array `arr`. The array accesses of each the four threads are marked by a different color. The version on the left ($n_{\text{jump}} = 1$) exhibits good temporal and spatial locality, i.e., there is reuse among the threads, and the data is accessed in coalesced fashion. When $n_{\text{jump}} = 6$ (right), there is no more reuse among the threads; such high values of $n_{\text{jump}} = 6$ also result in poor spatial locality (uncoalesced reads from global memory, which can be prohibitively expensive).

Algorithm 3 loess_main (Futhark-pseudocode)

```
1  loess_main([n]real x, [n]real y, [n]real w, long q,
2             [n_m]long ℓ_idx, [n_m]real λ_q, long n_jump) =
3    forall i = 0 to n_m - 1 do
4      real w[q], xw[q], x2w[q], fits_i[q], slopes_i[q]
5      forall j = 0 to q - 1 do
6        xx_j = x[ℓ_idx[i] + j] # q-slice
7        ww_j = w[ℓ_idx[i] + j] # q-slice
8        x̂_j = xx_j - min(i * n_jump, n - 1)
9        tmp1 = abs(x̂_j) / λ_idx[i]
10       tmp2 = 1.0 - tmp1 * tmp1 * tmp1
11       tmp3 = tmp2 * tmp2 * tmp2
12       # scale by the robustness weight
13       w[j] = tmp3 * ww_j
14       xw[j] = x̂_j * w[j]
15       x2w[j] = x̂_j * xw[j]
16     # compute symmetric matrix X for d = 1
17     a, b, c = sum(w), sum(xw), sum(x2w)
18     # compute matrix inverse using Cramer's rule
19     det = 1 / (a * c - b * b)
20     a1, b1, c1 = c * det, -b * det, a * det
21     # compute the matrix-vector product
22     forall j = 0 to q - 1 do
23       yy_j = y[ℓ_idx[i] + j] # q-slice
24       fits_i[j] = (w[j] * a1 + xw[j] * b1) * yy_j
25       slopes_i[j] = (w[j] * b1 + xw[j] * c1) * yy_j
26     fits[i], slopes[i] = sum(fits_i), sum(slopes_i)
27   return (fits, slopes)
```

of neighborhood weights, and (iii) the weighted OLS fitting. Based on the results of Table I, this kernel is responsible for $50-70\%$ of the total sequential compute time, making its optimization highly impactful for the overall performance. Pseudocode for **loess_main** is shown in Alg. 3:

The overall structure consists of a parallel map of size n_m (lines $3-26$), which contains inner-parallel operations of size q. The two outer maps were flattened into one map of size $m \cdot n_m$. The most straightforward approach to mapping the parallelism of the main LOESS procedure to the GPU is to execute the inner parallel constructs of size q sequentially. Then, each thread computes LOESS for a single $\hat{x}_j$. We reduce the memory footprint by not creating any buffers of size q in the main memory, and thus need to recompute the values of $\mathbf{y}$ and $\mathbf{w}$. For low values of n_{jump}, this results in a very fast implementation with excellent spatial and temporal locality. Figure 5 depicts the main trade-off between the value of n_{jump} and locality of reference. We call this the **outer-parallel** version.

The experiments from Figure 8 show that when n_{jump} reaches a device-specific threshold, the effects of uncoalesced access cannot be hidden anymore by caching and multithreading. Then, the performance of the main LOESS procedure deteriorates rapidly. We mitigate these effects by mapping the inner parallelism to the threads in a CUDA block. Now, consecutive threads access consecutive array elements (since the inner loop has stride 1), hence the reads are coalesced. In addition, this approach allows to store and reuse intermediate arrays from (fast) shared memory, thus saving two global-memory accesses per thread. This constitutes the intragroup version (**intragroup**).

If the value of q becomes too large for a CUDA-block size, the intragroup version becomes infeasible. Another way of achieving spatial locality is to flatten all parallelism, i.e., we transform the nested parallelism of the main LOESS procedure into flat parallelism of size $m \cdot n_m \cdot q$. The resulting implementation consists of two regular-segmented reductions [29]: one covering lines 5-17, the other lines 22-26. The performance improves for high values of q—due to efficient sequentialization and temporal reuse from the L2 cache. The Flattened Version (**flat**) is also used when the outer parallelism is insufficient to saturate the GPU.

We select between versions at runtime using device-specific threshold parameters determined by a custom one-time autotuning procedure: (1) the jump threshold models the locality of the outer-parallel version, (2) the q threshold checks if the inner parallelism fits in the CUDA block, and (3) the parallelism threshold models whether the outer parallelism is enough to utilize the GPU.

D. Parallel Asymptotics and Kernel Weights

Assuming that n_{jump}, n_{inner}, n_{outer} are constants and denoting $q = \max(q_s, q_t, q_l)$, the parallel asymptotics of STLBATCHED are $\mathcal{O}(mNq)$ work and $\mathcal{O}(q + \log N)$ span. The depth is due to the q-nearest neighbor search, and the work is due to LOESS, which performs N operations on vectors of length q for each time series (see Alg. 2).

Table I shows the weight in the total sequential runtime of each of the main stages of the STL algorithm on nine datasets. One can notice that some weights vary significantly, e.g., the runtime of median varies from 19% on B_3 to 0% on B_1, where it is not used.

	Temperature		Vegetation	
	Ours	Statsmodels	Ours	Statsmodels
Data loading	1290.9s	1224.0s	7.1s	7.7s
GPU transfer	32.1s	—	2.2s	—
STL	706.1s	>5 days $\cdot 240$	12.5s	61095.8s
Postprocessing	5176.5s	$\approx$5170s	2.3s	55.5s
Total	7218.5s	>5 days $\cdot 240$	24.1s	61160.0s

TABLE II: Runtime measurements for the global case studies: our implementation (STLBatched) vs. statsmodels. The post-processing time for the temperature experiment is high due to the statistical significance test carried out in order to mask out pixels in Figure 1. After more than five days of running the CPU version has not yet finished processing the first out of 240 chunks of the data set.

V. Experimental Results

In this section, we present two large-scale geospatial case studies, which demonstrate the effectiveness of our parallel implementation for global-scale analyses. As shown below, our implementation allows to run such global-scale experiments using commodity hardware with single GPUs instead of large CPU clusters. In addition to these two case studies, we also analyze the performance of the induced GPU kernels.

A. Hardware

The experiments are run on machines with the following hardware:

1) **NVIDIA A100** PCIe 40GB GPU with AMD EPYC 7532 CPU and 512GB RAM. Peak GPU memory bandwidth: 1555GB/s. We use the CUDA backend of Futhark. If not specified, this is the machine used for the experiments.
2) **AMD MI100** GPU with AMD EPYC 7532 CPU and 512GB RAM. Peak GPU memory bandwidth: 1229GB/s. We use the OpenCL backend of Futhark.

B. Global-Scale Geospatial Case Studies

1) Temperature Trend Analysis: To analyse trend-term trends of air temperatures at the global scale, we used ECMWF Reanalysis v5 (ERA5) global hourly air temperature data at 0.25-degree resolution from 1940 to 2022 (a 1036800×30295 data set) [22]. The q_s and q_t were set as 365 and 7301 respectively, representing a one-year and a 20-year window. Figure 1 shows the trend magnitude of mid-day air temperature. The spatial patterns align with previous studies [25], with the Arctic experiencing the strongest long-term increase in temperature. A closer look at Greenland reveals a rapid increase in temperature since the 1990s (Figure 2), also in line with recent studies [27]. In contrast, the long-term trend in Western Australia is decreasing, while the short-term trend indicates an increase in temperature since 2008 (Figure 7).

2) Vegetation Trend Analysis: Satellite NDVI data combined with season-trend decomposition are often used to analyze vegetation dynamics globally in an unsupervised manner [10], [48]. Changes in vegetation over time can be quantified by the *trend magnitude*, which, for a given time series, may be defined as the slope of the least-squares regression line through the trend component from a STL decomposition [2]. A positive trend magnitude indicates vegetation growth is increasing (i.e., greening up linked to better environmental conditions), while a negative trend magnitude indicates vegetation growth is decreasing (i.e., browning, often related to drought events) [28], [49]. A large-scale experiment computing the trend magnitude for global, bi-weekly NDVI data from 1 July 1981 to 16 December 2015 (a 9331200×828 data set) is shown on Figure 6.

3) Performance comparison on NVIDIA A100: Table II compares the runtime of our implementation with *statsmodels* [44]. A three-order-of-magnitude speedup is observed for the vegetation trend analysis. Such speed-up enables parameter tuning, data exploration, and unsupervised monitoring using STL decomposition for large data such as global satellite imagery. This contribution is urgently needed as calculations for the entire time series need to be repeated for every new satellite image since the AVHRR satellite [32] and others provide new satellite images on a daily basis.

The large smoothing factor for the temperature trend analysis makes it infeasible to obtain results for the CPU version. The data are chunked into 240 parts; after five days of running, the CPU version has finished processing 0 out of these 240 chunks. Extrapolating from this, by assuming one chunk takes at least 5 days to process, it would take several years to run this experiment on a single CPU core.

C. Performance Analysis

This section evaluates the performance of our implementation and demonstrates the effectiveness of the code version selection scheme. We report performance in terms of memory bandwidth in GB/s. However, rather than using the exact number of global-memory accesses performed by each code version, we (pessimistically) consider instead the minimal number of accesses that *must* be performed by any implementation—e.g., we treat prefix sum similar to `memcpy` and count one read and one write per element.

a) Main LOESS Procedure: Figure 8 shows that the performance of the three code versions—outer parallel, intra-group, and flat—is sensitive to both dataset and hardware characteristics. For evaluation, we calibrated the jump factors via synthetic datasets where the performance of the outer-parallel version degraded steeply: 16 and 5 for the A100 and MI100, respectively. The outer-parallel version was $> 10\times$ faster than `flat` in the first subfigure (A100, $n_{\text{jump}} = 1, q = 129$), but up to $9\times$ slower than `flat` in the last subfigure (MI100, $n_{\text{jump}} = 8, q = 32767$). For small values of n_{jump} and high values of q the **outer-parallel** and **flat** code versions, respectively, reach a memory throughput higher than the peak GPU bandwidth because of temporal reuse in the L2 cache. We conclude that each of the code versions is best suited for a class of datasets, and the autotuning scheme is able to reliably select the best version.

b) STL and Its Kernels: Table III lists the execution time (ms) and percentage of GPU peak bandwidth for STL-

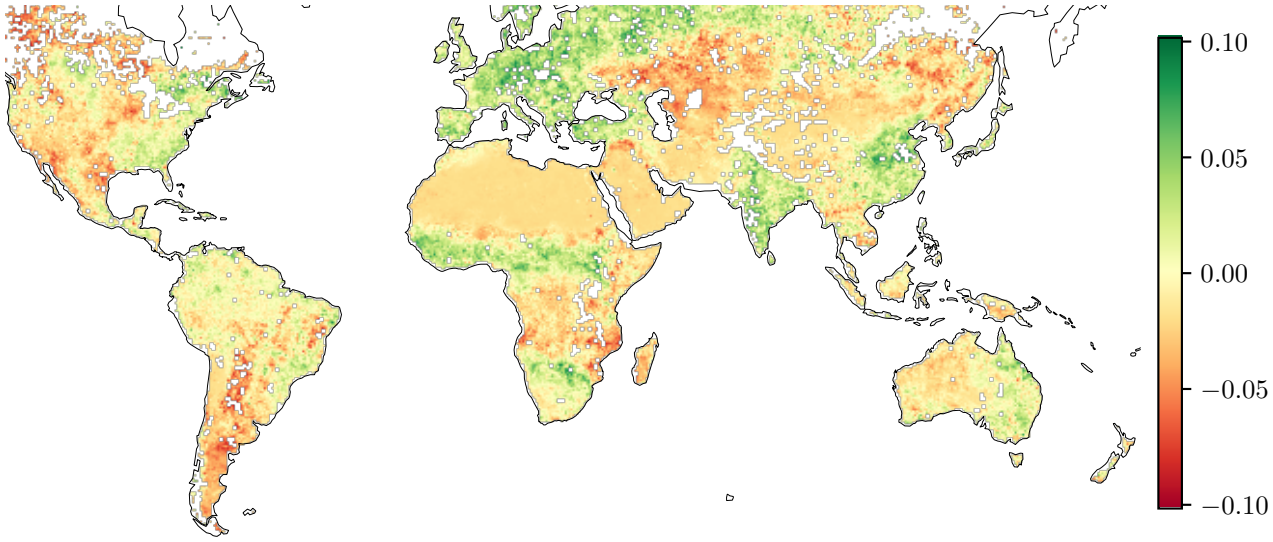


Fig. 6: Global vegetation trend magnitude (STL with $q_s = 25$). Values outside the 98th percentile are masked out (white).

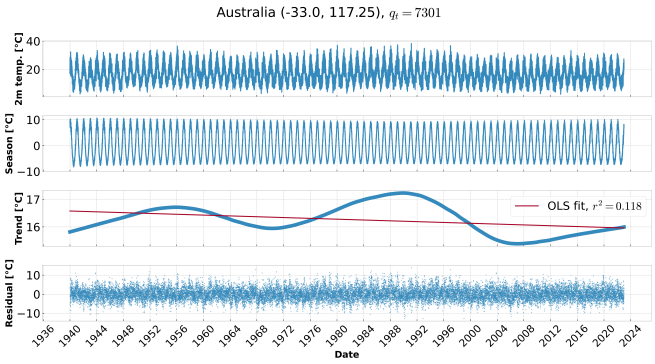


Fig. 7: A time series of temperatures measured for a single location in Australia and its STL decomposition.

	A1	A2	A3	B1	B2	B3	C1	C2	C3
loess_main	15.0	19.3	37.4	8.2	13.1	20.6	111.8	125.3	279.1
% max	88	68	88	88	55	87	207	184	207
nns	4.4	4.4	4.4	2.6	3.3	2.6	33.1	33.2	33.4
% max	61	61	61	59	46	59	166	166	165
interp	1.2	1.4	2.8	0.6	0.6	1.5	18.7	18.1	46.6
% max	104	93	113	122	122	122	131	135	131
filter	2.1	2.3	4.0	1.1	1.1	2.2	30.6	30.7	57.7
% max	61	55	63	65	66	65	74	74	74
ma	1.5	2.1	3.6	1.2	1.2	3.0	19.4	18.7	47.2
% max	66	49	71	47	48	47	100	104	103
median	0.0	0.0	16.2	0.0	0.0	19.7	0.0	0.0	162.6
STLBatched	25.2	29.8	83.0	15.3	17.5	68.0	254.7	270.1	842.1
% max	82	69	68	74	65	47	148	139	114

TABLE III: Execution time (ms) & percentage of peak memory bandwidth on A100.

Batched and its computational kernels as described in Sections IV-B and IV-C. The columns correspond to real-world datasets, which are configured as follows:

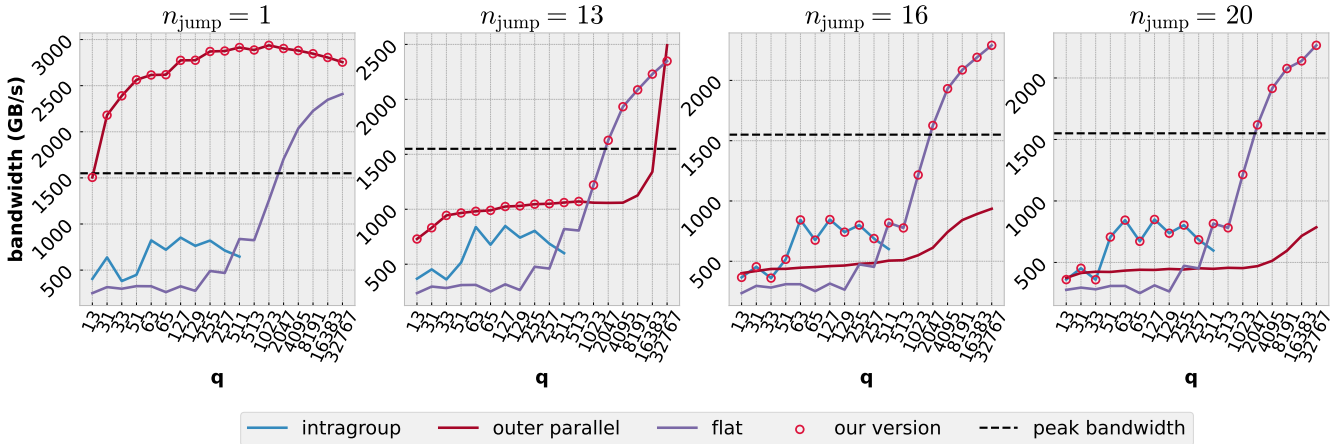
A_*: *Individual household electric power consumption* from the UCI Machine Learning Repository [12]. A dataset of size 4×2075259 with $n_{\text{jump}_s} = 36$, $n_{\text{jump}_t} = 217$, $n_{\text{jump}_l} = 145$, $q_s = 359$, $q_t = 2169$ and $q_l = 1441$.

B_*: *Beijing Multi-Site Air-Quality Data* from the UCI Machine Learning Repository [12]. A 132×35064 dataset with $n_{\text{jump}_s} = 5$, $n_{\text{jump}_t} = 27$, $n_{\text{jump}_l} = 17$, $q_s = 49$, $q_t = 261$ and $q_l = 169$.

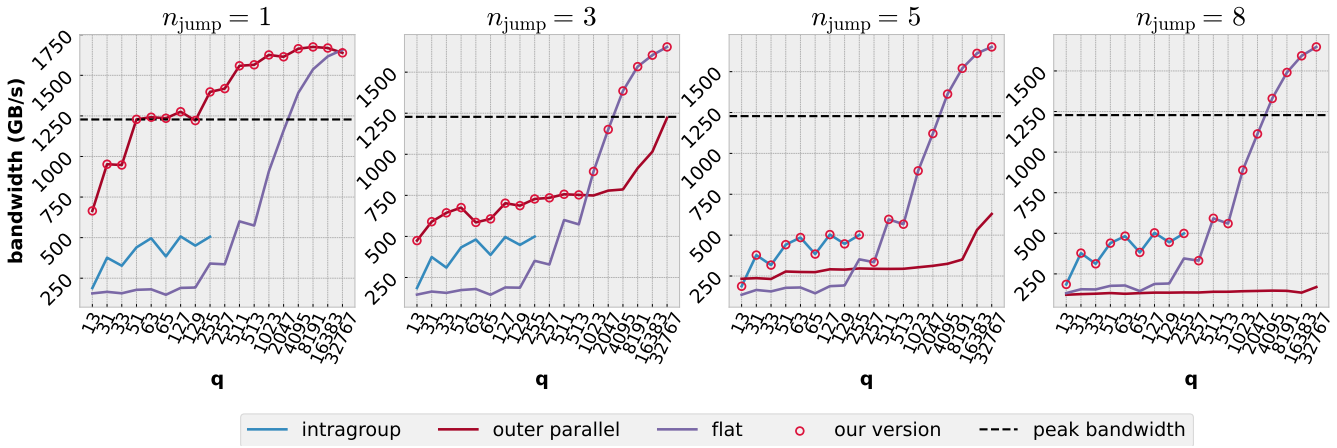
C_*: *NDVI time series* from the NOAA Advanced Very High Resolution Radiometer-derived Global Inventory Modeling and Mapping Studies (GIMMS) satellite image dataset [40]. A 186624×828 dataset with $n_{\text{jump}_s} = 3$, $n_{\text{jump}_t} = 4$ and $n_{\text{jump}_l} = 3$, $q_s = 25$, $q_t = 39$ and $q_l = 25$.

Each of the datasets is run in three different settings: the recommended setting by Cleveland *et al.* [8] ($*_1$), the LOESS-heavy setting that fits second-degree polynomials instead of first ($*_2$), and the robust setting, with 5 outer-loop iterations and median computation ($*_3$). The latter setup is only used in Table I and III. Most of the main computational kernels utilise around 50% or more of peak GPU bandwidth. The higher polynomial degree of the ($*_2$) setting incurs a slow down in the LOESS kernel across all three datasets. The bandwidth computation for the median kernel is omitted (in turn decreasing STL bandwidth slightly); median's performance is comparable with that of CUB's radix sort.

STLBatched reaches more than 100% of the peak GPU bandwidth for all C datasets since low values of n_{jump} are employed. This makes it possible to choose the outer parallel version of the main LOESS procedure, which exhibits good temporal locality (see Section IV-C4). For the B datasets, the implementation reaches no more than 74% since the intragroup version of LOESS is selected in two out of three applications for each dataset. The results are similar for the A datasets, where both the jump value and the value of q are high, which



(a) NVIDIA A100



(b) AMD MI100

Fig. 8: Performance of the main LOESS procedure on different GPU hardware with double precision evaluated on synthetic data. The x-axis displays the value of the smoothing parameter q, while the y-axis displays the memory bandwidth in GB/s. *Our version* corresponds to the code versions chosen by our selection scheme as described in Section IV-C4. *Peak bandwidth* shows the device-specific peak memory bandwidth. Results for the intragroup version stop at the maximum number of work-items per workgroup for the MI100, i.e., $q = 256$.

causes the flattened version of LOESS to be selected.

VI. Conclusion

This paper proposes a new GPU parallelization approach for the classic time series decomposition algorithm STL. Existing implementations of STL are impractical to apply over large datasets because they are not massively parallel and in some cases have quadratic memory complexity. To our knowledge, there are no known GPU implementations of STL and our work fills this gap. We demonstrate a one-time per-hardware auto-tuning procedure to select among alternative code versions for the main LOESS procedure, and experimentally show that our implementation can reach a significant fraction of the peak GPU bandwidth.

References

[1] Apaydin, H., Sattari, M.T., Falsafian, K., Prasad, R.: Artificial intelligence modelling integrated with singular spectral analysis and seasonal-trend decomposition using loess approaches for streamflow predictions. Journal of Hydrology **600**, 126506 (2021)

[2] Ben Abbes, A., Bounouh, O., Farah, I.R., de Jong, R., Martínez, B.: Comparative study of three satellite image time-series decomposition methods for vegetation change detection. European Journal of Remote Sensing **51**(1), 607–615 (2018)

[3] Bergmeir, C., Hyndman, R., Benítez, J.: Bagging exponential smoothing methods using stl decomposition and box–cox transformation. Int. J. of Forecasting **32**, 303–312 (04 2016)

[4] Bergmeir, C., Hyndman, R.J., Benítez, J.M.: Bagging exponential smoothing methods using stl decomposition and box–cox transformation. International journal of forecasting **32**(2), 303–312 (2016)

[5] Bovet, A., Makse, H.A.: Influence of fake news in twitter during the 2016 us presidential election. Nature communications **10**(1), 7 (2019)

[6] Cheng, G., Xie, X., Han, J., Guo, L., Xia, G.S.: Remote sensing image scene classification meets deep learning: Challenges, methods, benchmarks, and opportunities. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing **13**, 3735–3756 (2020)

[7] Clausen, M.T.: Regular Segmented Single-pass Scan in Futhark. Master's thesis, DIKU, University of Copenhagen (2021)

[8] Cleveland, R.B., Cleveland, W.S., McRae, J.E., Terpenning, I.: Stl: A seasonal-trend decomposition. J. Off. Stat **6**(1), 3–73 (1990)

[9] Cleveland, W.S.: Robust locally weighted regression and smoothing scatterplots. J. of the American Statistical Association **74**(368), 829–836 (1979)

[10] De Jong, R., Verbesselt, J., Zeileis, A., Schaepman, M.E.: Shifts in global vegetation activity trends. Remote Sensing **5**(3), 1117–1133 (2013)

[11] Ding, D., Zhang, M., Pan, X., Yang, M., He, X.: Modeling extreme events in time series prediction. In: International Conference on Knowledge Discovery & Data Mining (KDD). pp. 1114–1122. ACM SIGKDD (2019)

[12] Dua, D., Graff, C.: UCI machine learning repository (2017)

[13] Eklundh, L., Jönsson, P.: TIMESAT for Processing Time-Series Data from Satellite Sensors for Land Surface Monitoring, pp. 177–194. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-47037-5_9

[14] Fritsch, F.N., Carlson, R.E.: Monotone piecewise cubic interpolation. J. on Numerical Analysis **17**(2), 238–246 (1980), http://www.jstor.org/stable/2156610

[15] Gieseke, F., Rosca, S., Henriksen, T., Verbesselt, J., Oancea, C.E.: Massively-parallel change detection for satellite time series data with missing values. In: ICDE. pp. 385–396 (2020)

[16] Godfrey, L.B., Gashler, M.S.: Neural decomposition of time-series data for effective generalization. Transactions on Neural Networks and Learning Systems **29**(7), 2973–2985 (2018)

[17] Hafen, R.: stlplus. https://github.com/hafen/stlplus (2016)

[18] Hartman, J.D., Bakos, G.Á.: Vartools: A program for analyzing astronomical time-series data. Astronomy and Computing **17**, 1–72 (2016)

[19] Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning. Springer (2009)

[20] Henriksen, T., Hellfritzsch, S., Sadayappan, P., Oancea, C.: Compiling generalized histograms for GPU. In: Procs. of International Conference for High Performance Computing, Networking, Storage and Analysis. SC '20, IEEE Press (2020)

[21] Henriksen, T., Thorøe, F., Elsman, M., Oancea, C.: Incremental flattening for nested data parallelism. In: Procs. of Symposium on Principles and Practice of Parallel Programming. pp. 53–67. PPoPP '19, ACM, New York, NY, USA (2019), http://doi.acm.org/10.1145/3293883.3295707

[22] Hersbach, H., Bell, B., Berrisford, P., Hirahara, S., Horányi, A., Muñoz-Sabater, J., Nicolas, J., Peubey, C., Radu, R., Schepers, D., Simmons, A., Soci, C., Abdalla, S., Abellan, X., Balsamo, G., Bechtold, P., Biavati, G., Bidlot, J., Bonavita, M., De Chiara, G., Dahlgren, P., Dee, D., Diamantakis, M., Dragani, R., Flemming, J., Forbes, R., Fuentes, M., Geer, A., Haimberger, L., Healy, S., Hogan, R.J., Hólm, E., Janisková, M., Keeley, S., Laloyaux, P., Lopez, P., Lupu, C., Radnoti, G., de Rosnay, P., Rozum, I., Vamborg, F., Villaume, S., Thépaut, J.N.: The ERA5 global reanalysis **146**(730), 1999–2049. https://doi.org/https://doi.org/10.1002/qj.3803, https://rmets.onlinelibrary.wiley.com/doi/abs/10.1002/qj.3803

[23] Hewamalage, H., Bergmeir, C., Bandara, K.: Recurrent neural networks for time series forecasting: Current status and future directions. International Journal of Forecasting **37**(1), 388–427 (2021)

[24] Hoare, C.A.R.: Algorithm 65: Find. Comm. ACM. **4 (7)**, 321–322 (1961). https://doi.org/doi:10.1145/366622.366647

[25] IPCC: Climate Change 2021: The Physical Science Basis. Contribution of Working Group I to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change, vol. In Press. Cambridge University Press (2021). https://doi.org/10.1017/9781009157896

[26] Ivezić, v., et al.: LSST: From science drivers to reference design and anticipated data products. CoRR (2008)

[27] Jiang, S., Ye, A., Xiao, C.: The temperature increase in greenland has accelerated in the past five years. Global and Planetary Change **194**, 103297 (Nov 2020). https://doi.org/10.1016/j.gloplacha.2020.103297, https://doi.org/10.1016/j.gloplacha.2020.103297

[28] de Jong, R., Schaepman, M.E., Furrer, R., De Bruin, S., Verburg, P.H.: Spatial relationship between climatologies and changes in global vegetation activity. Global change biology **19**(6), 1953–1964 (2013)

[29] Larsen, R.W., Henriksen, T.: Strategies for regular segmented reductions on gpu. In: SIGPLAN Int. Workshop on FHPC. pp. 42–52. ACM (2017)

[30] Li, D., Chen, D., Jin, B., Shi, L., Goh, J., Ng, S.K.: Mad-gan: Multivariate anomaly detection for time series data with generative adversarial networks. In: Int. Conf. on Artificial Neural Networks (ICANN). pp. 703–716. Springer (2019)

[31] Li, S., Dragicevic, S., Castro, F.A., Sester, M., Winter, S., Coltekin, A., Pettit, C., Jiang, B., Haworth, J., Stein, A., Cheng, T.: Geospatial big data handling theory and methods: A review and research challenges. ISPRS Journal of Photogrammetry and Remote Sensing **115**, 119–133 (2016)

[32] Liu, H., Gong, P., Wang, J., Wang, X., Ning, G., Xu, B.: Production of global daily seamless data cubes and quantification of global land cover change from 1985 to 2020-imap world 1.0. Remote Sensing of Environment **258**, 112364 (2021)

[33] Maiti, M.: Applied Financial Econometrics. Palgrave Macmillan (2021)

[34] Marta, S.: Planet imagery product specifications. Planet Labs: San Francisco, CA, USA p. 91 (2018)

[35] Merrill, D.: Cuda unbound (cub) library. https://nvlabs.github.io/cub (2022)

[36] Merrill, D., Garland, M.: Single-pass parallel prefix scan with decoupled look-back. NVIDIA, Tech. Rep. NVR-2016-002 (2016)

[37] Munksgaard, P., Henriksen, T., Sadayappan, P., Oancea, C.: Memory optimizations in an array language. In: Int. Conference for High Performance Computing, Networking, Storage and Analysis. pp. 424–438. SC '22 (2022)

[38] Munksgaard, P., Breddam, S.L., Henriksen, T., Gieseke, F.C., Oancea, C.: Dataset sensitive autotuning of multi-versioned code based on monotonic properties. In: Zsók, V., Hughes, J. (eds.) Trends in Functional Programming. pp. 3–23. Springer International Publishing, Cham (2021)

[39] Pawlak, W.M., Hlava, M., Metaksov, M., Oancea, C.E.: Acceleration of lattice models for pricing portfolios of fixed-income derivatives. In: Procs of Int. Workshop on Libraries, Languages and Compilers for Array Programming. p. 27–38. ARRAY 2021, ACM (2021). https://doi.org/10.1145/3460944.3464309, https://doi.org/10.1145/3460944.3464309

[40] Pinzon, J.E., Tucker, C.J.: A non-stationary 1981–2012 avhrr ndvi3g time series. Remote sensing **6**(8), 6929–6960 (2014)

[41] Reichstein, M., Camps-Valls, G., Stevens, B., Jung, M., Denzler, J., Carvalhais, N., Prabhat: Deep learning and process understanding for data-driven earth system science. Nature **566**(7743), 195–204 (2019)

[42] Rojo, J., Rivero, R., Romero-Morte, J., Fernández-González, F., Pérez-Badia, R.: Modeling pollen time series using seasonal-trend decomposition procedure based on loess smoothing. Int. J. of biometeorology **61**(2), 335–348 (2017)

[43] Rojo, J., Rivero, R., Romero-Morte, J., Fernández-González, F., Pérez-Badia, R.: Modeling pollen time series using seasonal-trend decomposition procedure based on loess smoothing. Internat. Journal of Biometeorology **61**, 335–348 (2017)

[44] Seabold, S., Perktold, J.: statsmodels: Econometric and statistical modeling with python. In: 9th Python in Science Conference (2010)

[45] da Silva, R.G., Ribeiro, M., Mariani, V., Coelho, L.: Short-term forecasting of amazon rainforest fires based on ensemble decomposition model. CoRR **abs/2007.07979** (2020)

[46] Sun, T., Zhang, T., Teng, Y., Chen, Z., Fang, J.: Monthly electricity consumption forecasting method based on x12 and stl decomposition model in an integrated energy system. Math. Prob. in Engin. (2019)

[47] Verbesselt, J., Hyndman, R., Newnham, G., Culvenor, D.: Detecting trend and seasonal changes in satellite image time series. Remote Sensing of Env. **114**(1), 106–115 (2010)

[48] Verbesselt, J., Umlauf, N., Hirota, M., Holmgren, M., Van Nes, E.H., Herold, M., Zeileis, A., Scheffer, M.: Remotely sensed resilience of tropical forests. Nature Climate Change **6**(11), 1028–1031 (2016)

[49] Wanyama, D., Moore, N.J., Dahlin, K.M.: Persistent vegetation greening and browning trends related to natural and human activities in the mount elgon ecosystem. Remote Sensing **12**(13), 2113 (2020)

[50] Wen, Q., Gao, J., Song, X., Sun, L., Xu, H., Zhu, S.: Robuststl: A robust seasonal-trend decomposition algorithm for long time series. In: AAAI. pp. 5409–5416 (2019)

[51] Wen, Q., Zhang, Z., Li, Y., Sun, L.: Fast robuststl: Efficient and robust seasonal-trend decomposition for time series with complex patterns. In: KDD. pp. 2203–2213 (2020)

[52] Zhang, C., Song, D., Chen, Y., Feng, X., Lumezanu, C., Cheng, W., Ni, J., Zong, B., Chen, H., Chawla, N.V.: A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data. In: AAAI Conference on Artificial Intelligence. vol. 33, pp. 1409–1416 (2019)

[53] Zhu, Z.: Change detection using landsat time series: A review of frequencies, preprocessing, algorithms, and applications. ISPRS J. of Photogrammetry and Remote Sensing **130**, 370–384 (2017)