SPRINGALD: GPU-Accelerated Window-Based Aggregates Over Out-of-Order Data Streams

Gabriele Mencagli, Patrizio Dazzi, and Massimo Coppola

***Abstract*—An increasing number of application domains require high-throughput processing to extract insights from massive data streams. The Data Stream Processing (DSP) paradigm provides formal approaches to analyze structured data streams considered as special, unbounded relations. The most used class of stateful operators in DSP are the ones running sliding-window aggregation, which continuously extracts insights from the most recent portion of the stream. This article presents SPRINGALD, an efficient sliding-window operator leveraging GPU devices. SPRINGALD, incorporated in the WINDFLOW parallel library, processes out-of-order data streams with watermarks propagation. These two features—GPU processing and out-of-orderliness—make SPRINGALD a novel contribution to this research area. This article describes the methodology behind SPRINGALD, its design and implementation. We also provide an extensive experimental evaluation to understand the behavior of SPRINGALD deeply, and we showcase its superior performance against state-of-the-art competitors.**

***Index Terms*—Data stream processing, window-based aggregates, out-of-order data streams, GPUs.**

I. INTRODUCTION

COMPUTING complicated analytics over high-velocity data streams is a challenging topic having increasing importance in domains such as IoT applications, social media analysis, log monitoring, financial markets, and many others. *Data Stream Processing* [1] (DSP) is an approach to query data streams proposed years ago by the database community, with a dialect of SQL (e.g., the Continuous Query Language [2]) to process streams like finite relations. This approach adapts the semantic foundation of some stateful relational operators—mainly aggregates and joins—to unbounded streams by introducing the notion of *window* [3]. Motivated by the idea that only the most recent data are relevant, such operators repeatedly compute results over independent and dynamic windows of the stream representing the last received inputs (called *tuples* hereinafter).

Manuscript received 5 March 2024; revised 15 July 2024; accepted 17 July 2024. Date of publication 22 July 2024; date of current version 31 July 2024. This work was supported in part by the Italian PRIN project OUTFIT under Grant 2022BAL2F3, and in part by NOUS (A catalyst for EuropeaN ClOUd Services in the era of data spaces, high-performance and edge computing) HORIZON-CL4-2023-DATA-01-02 project, G.A. under Grant 101135927. Recommended for acceptance by F. Zhang. *(Corresponding author: Gabriele Mencagli.)*

Gabriele Mencagli and Patrizio Dazzi are with the Department of Computer Science, University of Pisa, 56126 Pisa, Italy (e-mail: gabriele.mencagli@unipi.it; patrizio.dazzi@unipi.it).

Massimo Coppola is with ISTI-CNR, 56127 Pisa, Italy (e-mail: massimo.coppola@isti.cnr.it).

Digital Object Identifier 10.1109/TPDS.2024.3431611

The first generation of *Stream Processing Engines* (SPEs) was proposed 20 years ago. They were called *Data Stream Management Systems* (DSMSs) (e.g., Aurora [4], Borealis [5] and others) to highlight their direct evolution from traditional Data Base Management Systems. They provided full support to CQL and stateful operators, and they dealt with several stream imperfections like stream disordering and duplicates [6]. Modern SPEs (e.g., Storm [7], Flink [8]) supersede DSMSs by supporting different kinds of streams, both structured and unstructured.

Sliding-window aggregation is one of the main stateful operators of relational stream processing, which usually affects the overall query performance [3]. It computes a binary aggregation function over all tuples belonging to the same window, which usually captures the last data received. Modern SPEs provide built-in operators to perform aggregation over *count-* and *time-based windows*. For example, the built-in algorithm adopted by Flink is based on the *bucket-per-window* approach [9], where each open window (bucket) is associated with a partial result, and upon arrival of a new tuple all buckets affected by that tuple are considered and their results updated. To enforce potential overlapping between consecutive windows, several optimized algorithms have been proposed over the years (they will be reviewed in Section VII). Examples of such techniques are panes [10], pairs [11], and other general incremental aggregation approaches [12], [13], [14], [15]. When the data stream may be received out-of-order—with tuples having non-monotonically increasing timestamps—sliding-window aggregation requires special care to close windows properly without missing data [6], [16].

As modern hardware is inherently parallel (i.e., composed of more multi-core CPUs and co-processors such as GPUs), its exploitation to speed up query performance has been the subject of prior works (see Section VII). StreamBox [17], LightSaber [18], Grizzly [19] are some relevant papers focusing on approaches starting from a CQL-based declarative program and exploiting multi-core CPUs efficiently. SABER [20] and FineStream [21] are two SPEs targeting heterogeneous systems based on CPUs and GPUs. Although with relatively high performances, they support GPU processing for sliding-window aggregation by assuming *in-order* streams, i.e., the stream exhibits monotonically increasing timestamps. While in-order streams can be realistic in some use cases, out-of-order streams are the norm, especially when events originate from several sources displaced across a wide area network.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This paper presents SPRINGALD,[1] a GPU-accelerated sliding-window aggregator supporting associative and commutative aggregation functions, and working with out-of-order data streams, i.e., data streams whose items are watermarked, i.e., associated with metadata indicating the progress of the stream. Both aspects, i.e., *GPU exploitation* and *out-of-orderliness*, synergically studied and applied in SPRINGALD, represent a novel contribution to the field. Our approach processes tuples with a two-staged procedure: first, tuples are processed out-of-order to generate partial results; second, when some partial results are considered complete (i.e., based on the last received watermark), they are added to a binary tree structure representing all the partial aggregates useful to compute a set of contiguous windows of the same stream. The algorithmic approach underpinning SPRINGALD is inspired by some techniques originally introduced for sequential processing (i.e., pane-based aggregation [10] and binary trees [12]), which we have implemented efficiently on GPU hardware accelerators with SIMD (*Single-Instruction Multiple-Data*) processing capabilities. Such a design provides high throughput with count- and time-based windows and stream disordering.

The contributions of this paper are the following:

- SPRINGALD employs a pipelined architecture composed of two stages named PLS (Pane-level Stage) and WLS (Window-level Stage). This architecture, inspired by the original work about panes [10], has been revisited in this paper to parallelize each stage on GPU. Furthermore, the pipeline approach allows hiding latencies such as host-to-device and device-to-host copies;
- The parallelization of PLS adopts an approach where panes are aggregated on the fly by processing input batches in parallel on GPU, and by keeping a circular buffer of pending panes that cannot be considered complete yet. So, we succeed in incorporating an out-of-order processing logic on the device, where panes are considered closed once a watermark closing them is available from the input stream;
- The parallelization of WLS is our second contribution. Inspired by some previous work adopting binary aggregation trees [12], we design a new data structure called *Batched Pane Aggregation tree* (B-PAT). Differently from previous works, the B-PAT can be configured to have a width in terms of leaves sufficient to compute $N_w \geq 1$ consecutive windows, so storing partial aggregates that contribute to more windows of the stream. This increases data parallelism opportunities since windows can be computed in parallel on the device with high bandwidth. Furthermore, we design the B-PAT to have a pointer-less and flat layout particularly efficient for being updated by GPU hardware.

The paper proposes also a reasoned comparison of SPRINGALD against SABER [20], a GPU-accelerated SPE for in-order streams, and SCOTTY [22], a CPU-based out-of-order aggregation framework integrated into modern SPEs.

The outline of the paper is the following. Section II introduces background concepts. Sections III and IV provide an overview of SPRINGALD and its design. Section V shows the implementation. Section VI outlines the experimental evaluation, Section VII introduces related works, and Section VIII draws the conclusions.

[1] Springald was a medieval torsion artillery device for throwing bolts.

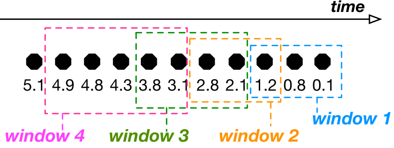

Fig. 1. Time-based windows with $w = 2$ and $s = 1$.

II. BACKGROUND AND MOTIVATION

We briefly introduce the essential concepts of data stream processing and out-of-order streams used in the paper, and we show the motivations of our work.

Data Stream Processing DSP originates from the database community [1]. It allows the processing of data streams to extract online analytics and complex events. DSP queries are composed of *operators* executing intermediate transformation stages and connected in data-flow graphs. Operators can be *stateless*—computing pure functions whose outputs depend solely on tuples—or *stateful*, which maintain and update a set of data structures while processing tuples.

Window-Based Aggregates Since streams are unbounded, statistics like aggregates (e.g., max, min, average, count) are computed over subsets of tuples called *windows*. Depending on how tuples are grouped into windows, different models have been proposed over the years [23]. In this paper, we consider count- and time-based windows. They are both expressed with two parameters: the window *length* w and the *slide* s, where consecutive windows contain overlapped data if $s < w$. In time-based windows, w and s are expressed in time units, while in count-based windows in number of tuples. Fig. 1 shows an example of time-based windows with $w = 2$ and $s = 1$ (seconds). Tuples are associated with timestamps. The tuple with timestamp 2.8 belongs to both windows 2 and 3 in the figure.

Out-of-Order Data Streams Data streams are considered *ordered* if the timestamps of the received tuples are monotonically increasing. If t_i is the i-th received tuple, and $t_i.ts$ is its timestamp, we say that t_i is a *late tuple* if $\exists\, t_j$ with $j < i$ such that $t_j.ts > t_i.ts$. In the presence of late tuples, so with out-of-order data streams, detecting the ending of time-based windows is more challenging. With $w = 5\,sec$ and $s = 1\,sec$, if the stream is ordered (no late tuples exist), the first window is complete when the first tuple having a timestamp greater or equal to 5 is received. Unfortunately, this is no longer true with out-of-order data streams. A solution is to associate with each tuple t_i a further attribute called *watermark* ($t_i.wm$ s.t. $t_i.wm \leq t_i.ts$) of the same type of the timestamp. Such a watermark indicates that $\forall\, t_j\; s.t.\, j > i,\; t_j.ts > t_i.wm$, i.e., all future tuples received after t_i will have timestamps greater or equal than $t_i.wm$. In addition to being conveyed by the tuples as further attributes, watermarks can be propagated through dedicated meta tuples called *punctuations*. Watermarks, which have to be monotonically increasing, are often calculated by the sources in a

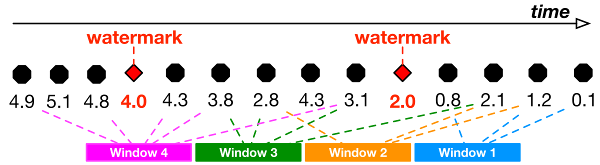

Fig. 2. Out-of-order stream with watermarks.

user-defined manner. Fig. 2 shows an example of an out-of-order data stream by referring to the same tuples, timestamps, and windows of Fig. 1. We suppose two punctuations (red diamonds) with watermarks of 2.0 and 4.0 respectively. Window 1 is finalized when the first watermark is received, while windows 2 and 3 at the time instant when the second watermark arrives.

Motivation Using GPU hardware to accelerate DSP workloads is compelling and challenging at the same time. Traditional database operators can greatly benefit from GPU acceleration and SIMD execution. This has been proved true for specialized sorting algorithms and joins, but also for projection and selection that are embarrassingly parallel [24]. Since DSP operators originated from linear algebra and database theory, DSP workloads may benefit from GPU acceleration too. However, DSP poses additional challenges due to the transient nature of the data that is continuously *in motion* (i.e., not persistently available in advance). This requires data streams to be bucketed in batches of tuples with a linear layout in device memory, which is transferred to the device continuously so that the GPU is never idle. Furthermore, since not all the fields of the tuples might be useful for the offloaded computation, projections, and data lifting are of special importance to save host-to-device memory bandwidth (e.g., by keeping only the fields useful to compute the aggregation function). Last but not least, DSP operators are often complex due to the windowing nature of the computation, where portions of the stream with dynamic sizes, and potential overlaps, are continuously aggregated. The amount of data parallelism is often large for those operators (e.g, a window aggregate is a reduction over a linear array of tuples), which however might be not fully exploited if the dynamic content of windows is implemented with continuous allocation/deallocation in device memory and redundant data transfers. SPRINGALD aims at providing an efficient solution to such problems.

III. SPRINGALD OVERVIEW

SPRINGALD is part of the WINDFLOW library [25]. This section provides an overview of WINDFLOW and SPRINGALD.

A. Data Pipelining With WINDFLOW

WINDFLOW is a C++17 library[2] enabling DSP on multicores and GPUs. It is based on the composition of a few components called *building blocks*, which allows building graphs of operators communicating through Single-Producer Single-Consumer (SPSC) lock-free queues [26].

[2]WINDFLOW and SPRINGALD are available in GitHub at https://github.com/ParaGroup/WindFlow.

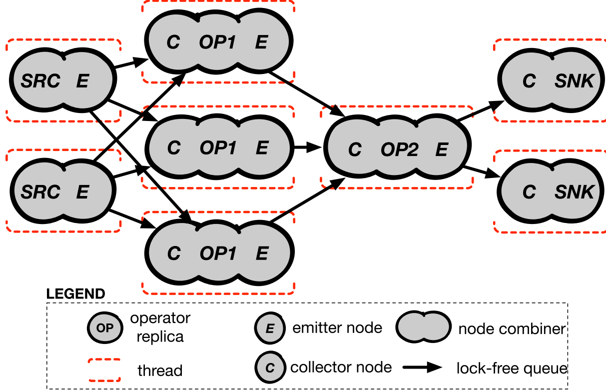

Fig. 3. A WINDFLOW data-flow graph composed of a Source, two intermediate operators, and a Sink.

Fig. 3 shows a data-flow graph incorporating a Source (SRC), a Sink (SNK), and two internal operators (OP1 and OP2). Each operator has its own number of replicas to express data parallelism. The graph is obtained by nesting all-to-alls, pipelines, combiners (applying fusion of sequential functions), and nodes recursively according to the regular grammar in [25]. In addition to nodes implementing operators (OP) that execute user-defined functions, the graph is further composed of nodes added by the run-time system transparently to the user: they are *emitter* (E) and *collector* (C) nodes performing data distribution policies (e.g., load balancing, key partition) and data gathering strategies.

SPRINGALD is a novel operator incorporated in WINDFLOW, which can be used at any point of the dataflow to provide efficient massive window aggregation. The user is requested to provide two device lambdas (i.e., lambda runnable on the GPU). The first function $\mathsf{lift}(\mathsf{t} : \mathsf{Input_t}) : \mathsf{Agg_t}$ converts an input into an aggregate of type $\mathsf{Agg_t}$. The second function $\mathsf{combine}(\mathsf{a1} : \mathsf{Agg_t}, \mathsf{a2} : \mathsf{Agg_t}) : \mathsf{Agg_t}$ is associative and commutative, and combines two aggregates into one resulting aggregate value.

B. SPRINGALD Approach

SPRINGALD combines two existing techniques. The first is pane-based processing [10], where the stream is logically partitioned into disjoint panes of length equal to $p = gcd(w, s)$, where w, s are the window length and slide respectively. By computing a result per pane (so aggregating all the tuples falling in the pane boundaries), we can reuse pane results shared by multiple consecutive windows. However, if p is small (e.g., if the slide parameter is small), this two-level aggregation approach is not effective since the number of computations per window is still proportional to w. For this reason, we integrate pane-based processing with aggregation trees [12] (second technique), where leaves are pane results and internal nodes are partial aggregates built on top of pane results. So doing, internal nodes can be reused to compute results of consecutive windows without recomputing them from scratch, requiring a logarithmic number of computations per window at the worst case.

Although these two techniques are used in the literature, and in recent CPU implementations [22], we provide a novel design and

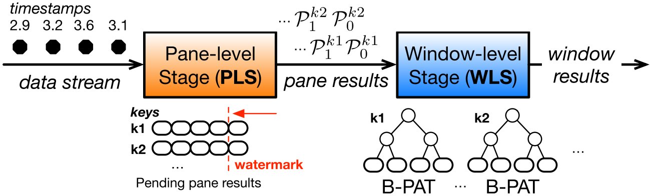

Fig. 4. General design of SPRINGALD with PLS and WLS.

implementation able to address two important issues: *i)* the offloading of the pane processing and tree maintaining procedures on the GPU; *ii)* the handling of out-of-order data streams, which is implemented directly on GPU with minimal host intervention. The three pillars of the SPRINGALD methodology are:

Overlapped data transfers: SPRINGALD makes use of properly designed emitter functionalities (Section III-A) capable of overlapping the copies of a new batch in the device memory with the preparation of the next batch with new data received from the Sources. This allows masking the copy overhead and exploits the available host-to-device memory bandwidth at best.

Out-of-order panes aggregation on GPU: Out-of-order data streams require that several pane results cannot be considered closed and must be kept in memory until a proper watermark closes them. We engineer a new approach to keep pane results in device memory in a circular buffer. When new batches are received, they are pre-processed on GPU to pre-aggregate tuples falling in the same panes. Then, those partial results are aggregated with existing pane results in the circular buffer efficiently and in parallel using the GPU.

GPU-oriented aggregation tree: Pane results that are complete upon a new watermark arrival are copied by overwriting a subset of the leaves of the tree. Our novel tree structure, called *Batched Pane-aggregation Tree*, has a size-independent from the specification of w and s and is based on a parameter N_w indicating how many consecutive windows can be computed using the leaves and internal nodes of the tree. This allows higher opportunities to exploit data parallelism, since GPU processing is used both for updating the tree upon new pane results arrival, as well as to compute in parallel N_w windows using the internal nodes.

In the next section, we provide an algorithmic description of SPRINGALD.

IV. SPRINGALD DESIGN

From a structural viewpoint, SPRINGALD is a pipeline of two stages named *Pane-level stage* (PLS) and *window-level stage* (WLS), as in Fig. 4.

The two stages try each to solve the two main issues that we considered in our work: the efficient processing of window aggregates on GPU, and the management of out-of-order data streams. The first problem is addressed mostly by the design of the WLS. It keeps a proper aggregation tree where each leaf corresponds to a complete pane result. Let w be the window length and s the slide parameter, we define the *pane length* as $p = gcd(w, s)$, so panes are non-overlapped and adjacent, and each tuple belongs to one pane only. The result of a pane is the aggregate computed over all tuples falling in the pane boundaries. Every time a new pane is complete (no future tuple falling in its boundaries is expected to arrive), the corresponding leaf is updated and all the internal nodes affected by the change are updated too. If a given number of new panes are completed, the tree will be used to compute the final aggregates of a set of contiguous windows without recomputing them from scratch, so using partial aggregates corresponding to a portion of tuples shared by consecutive windows.

Algorithm 1. PLS Logic to Process a New Input.

1 **Function** `processInput`(*input_t t*)
2 $\quad id \leftarrow t.ts/p$
3 $\quad k \leftarrow key_extractor(t)$
4 $\quad$ **if** *H[k] = nil* **then**
5 $\quad\quad H[k] \leftarrow$ create new $\mathcal{Q}$
6 $\quad\quad H[k].first \leftarrow 0, H[k].last \leftarrow id$
7 $\quad\quad H[k].append(id + 1, \perp)$
8 $\quad$ **else if** $H[k].last < id$ **then**
9 $\quad\quad H[k].append(id - H[k].last, \perp)$
10 $\quad\quad H[k].last \leftarrow id$
11 $\quad H[k][id - H[k].first] \leftarrow H[k][id - H[k].first] \oplus t.val$
12 $\quad wp \leftarrow t.wm/p - 1$ `// last complete pane id`
13 $\quad$ **if** $wp \geq 0$ **then**
14 $\quad\quad$ **for** $i \leftarrow H[k].first$ **to** wp **do**
15 $\quad\quad\quad$ **send** $H[k][i - H[k].first]$ **to** WLS
16 $\quad\quad H[k].erase(wp - H[k].first + 1)$
17 $\quad\quad H[k].first \leftarrow wp + 1$
18 $\quad\quad H[k].last \leftarrow \max\{wp + 1, H[k].last\}$

Unfortunately, the presence of out-of-order streams poses additional challenges. The tree structure must have a fixed size to be efficiently stored in GPU memory. However, with out-of-order streams, the number of active panes is not predictable since it depends on the stream delay. The PLS addresses this problem by keeping a dynamic set of pending pane results in device memory, aggregating them efficiently as new batches of tuples are made available by data sources, and by moving complete panes when watermarks arrive at the WLS (where more leaves of the tree will be updated triggering the computation of a new set of windows that are now surely complete).

A. Pane-Level Stage

PLS receives a disordered stream of tuples and aggregates them into pane results as in Algorithm 1. It keeps a hash table H mapping each key onto a container $\mathcal{Q}$. This container stores non-complete pane results of the given key attribute (i.e., the ones that cannot be considered complete according to the last received watermark). The container allows different operations: *i)* the access to a pane result at position i; *ii)* append(n, v) to append $n > 0$ new pane results with the same initial value v at the end; *iii)* erase(n) to delete the first $n > 0$ pane results from the beginning. For each key k, $H[k]$ keeps the identifiers of the first pane result (first) and of the last one last stored in the container. So, accessing position i of the container corresponds to the pane result with identifier first$+i$.

The stage extracts the pane identifier corresponding to the timestamp of the current tuple t, and its key attribute (lines 2-3). Lines 4-7 handle the arrival of the first tuple of a given key, while lines 8-10 append new pane results to the container. The

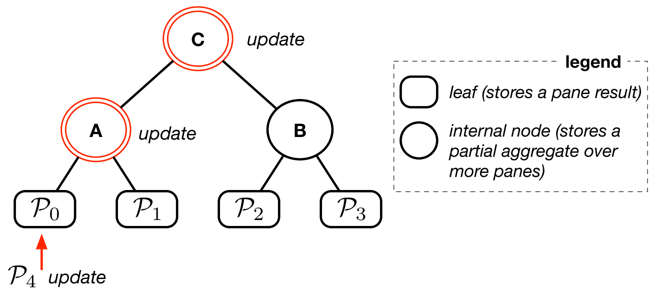

Fig. 5. Size-4 PAT built over time-based windows with $w = 8$, $s = 2$ and pane length $p = 2$ time units.

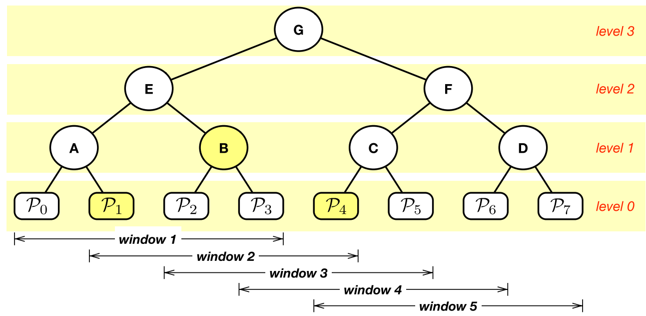

Fig. 6. Size-8 B-PAT over time-based windows with $w = 4$, $s = 1$, $p = 1$ time units, and $n_w = 5$ windows.

value $\perp$ is the neutral element of the binary operator $\oplus$ used for the aggregation. Line 11 applies the update of the pane result corresponding to the received tuple (we use the attribute val to compute the aggregate). Lines 12-18 describe the actions to deliver complete pane results to WLS. This happens by using the watermark conveyed by the tuple, which determines the identifier of the last pane that can be considered complete (wp in the code). Finally, the pane results transmitted to WLS are removed from the container, and first and last are updated.

B. Window-Level Stage

This stage receives complete pane results and maintains a binary tree for each existing key. Of such a tree, the leaves are pane results, and intermediate nodes are partial aggregates built on top of pane results. This structure is similar to the FlatFAT proposed by Tangwongsan et al. [12]. However, while the FlatFAT structure was originally proposed for count-based windows, our approach is suitable to represent time-based windows too. Before introducing the definition of this tree, we first present a simplified version of it called *Pane Aggregator Tree* (PAT).

1) PAT: We assume time-based windows of length w, slide s, and pane length $p = gcd(w, s)$ time units. Let n be the number of panes of each window, so $n = w/p$. A size-n PAT for an associative binary operator $\oplus : \mathcal{D} \rightarrow \mathcal{D}$ is a complete binary tree having $n > 0$ leaves denoted by $\mathcal{L}(0), \mathcal{L}(1), \ldots, \mathcal{L}(n-1)$, where $\mathcal{L}(i) \in \mathcal{D}$ for $i = 0..n-1$. For the moment, we assume that n is a power of two. Fig. 5 shows the idea of a size-4 PAT applied to time-based windows with $w = 8$, $s = 2$ and $p = gcd(8, 2) = 2$. In this part, we assume that panes results coming from PLS all belong to the same key, so WLS keeps one tree structure only.

The initial tree is built over the first n panes of the stream. Leaf $\mathcal{L}(i)$ contains the pane result $\mathcal{P}_i \in \mathcal{D}$ computed over the tuples t whose timestamps $t.ts$ fall inside the boundary of the i-th pane, i.e., $t.ts \in [i \cdot p, (i+1) \cdot p)$. For each internal node v, it contains a partial aggregate $\mathcal{T}(v) \in \mathcal{D}$ such that $\mathcal{T}(v) = \mathcal{T}(\mathsf{left}(v)) \oplus \mathcal{T}(\mathsf{right}(v))$, where $\mathsf{left}(v)$ and $\mathsf{right}(v)$ represent the left and right child of v respectively. By using simple mathematical induction and the associativity of $\oplus$, the root of the tree v stores the aggregate computed over all the panes of the first window, i.e., $\mathcal{T}(v) = \mathcal{P}_0 \oplus \mathcal{P}_1 \oplus \cdots \oplus \mathcal{P}_{n-1}$.

The procedure to initialize the tree requires computing each one of the $n-1$ internal nodes, so with $O(n)$ complexity. The aggregate over the whole window is stored in the root and can be retrieved by the user in $O(1)$. When s/p new pane results are made available by PLS (one in the example), they will be stored in the tree by replacing the oldest s/p leaves circularly, and by updating the internal nodes affected by the changes. Fig. 5 shows this example, with the first $s/p = 1$ leaves updated by replacing P_0 with P_4. This change requires updating the internal nodes A and C. In general, with s/p updates of the leaves, we need $O(s/p + \log_2(w/s) \cdot s/p)$ calls of $\oplus$ to update the internal nodes, as demonstrated in [12].

2) B-PAT: The *Batched PAT* (B-PAT) is a generalization of PAT to incorporate partial results useful to compute $n_w > 0$ consecutive windows. This allows the B-PAT to be composed of a sufficient number of internal nodes, so exposing high parallelism regardless of the size of the windows which is an application-dependent parameter.

A size-m B-PAT for an associative binary operator $\oplus : \mathcal{D} \rightarrow \mathcal{D}$ is a complete binary tree having $m = w + (n_w - 1) \cdot s$ leaves. For the moment being, we assume w, s, and n_w such that m results in a power of two. Fig. 6 shows an example of a size-8 B-PAT for time-based windows with $w = 4$, $s = 1$, $p = gcd(w, s) = 1$, and $n_w = 5$ consecutive windows.

The initialization of the B-PAT is analogous to the PAT. The main difference is that the computation of the partial aggregates stored in the internal nodes can be limited up to level 2 in the example of Fig. 6, since the node E stores the result of the first window, while F the one of the fifth window, whereas the root would contain an aggregate spanning more consecutive windows, so it is never used. In general, the initialization procedure of a size-m B-PAT initializes all the internal nodes at level $l \leq \log_2(n)$, with $n = w/p$. Similarly to the PAT, when the right amount of pane results are made available by PLS, such new results will be stored in the same number of leaves storing the oldest pane results, which requires updating some internal nodes up to level $l \leq \log_2(n)$, as previously discussed. We denote the number of pane results needed to update the tree and to start computing the next n_w windows as s_b, which is equal to $(n_w \cdot s)/p$. So, increasing n_w increases the number of complete pane results to receive from PLS to start the update procedure of the B-PAT.

We focus on how to compute the final aggregate of a given window in the B-PAT. To compute the aggregate over the second

Algorithm 2. Functions Prefix and Range.

```
1  Function prefix(v, k) // v: root, k: number of
     leaves of the k-th prefix
2    l ← level(v)
3    if 2^l = k then
4      return T(v)
5    else
6      if 2^(l−1) ≥ k then
7        return prefix(left(v), k)
8      else
9        return T(left(v)) ⊕ prefix(right(v), k − 2^(l−1))
10 Function range(v, i, j) // v: root, i: position first
     leaf, j: position last leaf
11   l ← level(v)
12   if i > j then
13     return range(v, i, 2^l − 1) ⊕ range(v, 0, j)
14   num ← j − i + 1
15   if 2^l = num then
16     return T(v)
17   else
18     if 2^(l−1) ≤ i then
19       return range(right(v), i − 2^(l−1), j − 2^(l−1))
20     else if 2^(l−1) > j then
21       return range(left(v), i, j)
22     else
23       return suffix(left(v), 2^(l−1) − i) ⊕
           prefix(right(v), j − 2^(l−1) + 1)
```

window in Fig. 6, we compute $\mathcal{P}_1 \oplus B \oplus \mathcal{P}_4$, so re-using the partial aggregate stored in one internal node to reduce the number of $\oplus$ calls. To generalize this procedure, we introduce some definitions. Let the *k-th prefix* be the aggregate computed over the $k > 0$ leftmost leaves of the tree, while the *k-th suffix* is the aggregate over the $k > 0$ rightmost leaves of the tree. The algorithm to compute a prefix is in Algorithm 2 (lines 1-9). To compute suffixes, we can use a call to suffix instead of prefix at line 7, and compute the result of the aggregation between the result of the recursive call of suffix applied to left(v) and the partial result stored in right(v) (line 9).

These two algorithms are used by range(v, i, j), which is the aggregate computed considering the leaves from $\mathcal{L}(i)$ to $\mathcal{L}(j)$ where v is the root. For example, range$(v, 2, 5)$ corresponds to the result of the third window in Fig. 6. Although in Fig. 6 all windows in the B-PAT have $i < j$, it is possible that windows wrap around. In the example of Fig. 6, $\mathcal{P}_8$ and $\mathcal{P}_9$ are stored in the first two leaves from the left, and the aggregate of the sixth window is computed by calling range$(v, 6, 2)$. As shown in lines 12-13, the result is computed in this case by calling range recursively in the sub-trees rooted at the left and right children respectively.

Theorem 4.1. The range procedure described in Algorithm 2, applied over a size-m B-PAT, takes $O(\log(m))$ time.

Proof. Prefix traverses the tree following a root-to-leaf path. At each level, it performs one $\oplus$ call at most. So the procedure, as well as suffix that is symmetric, has logarithmic complexity. The range procedure still traverses the tree top-down, finding the least common ancestor (LCA) between $\mathcal{L}(i)$ and $\mathcal{L}(j)$. Once found, the procedure calls suffix and prefix on the sub-trees rooted at the left and right child respectively (line 23), which have logarithmic complexity, as seen. So, the number of $\oplus$ calls by range is logarithmic in the number of leaves. □

Arbitrary Window Parameters B-PAT is a complete tree, so the number of leaves is a power of two. However, if w, s, and n_w are arbitrary values, m might not be a power of two. In that case, we build a size-m' B-PAT where m' is the smallest power of two greater or equal than m. Therefore, before initializing the tree, we fill the unused leaves with a marker $\perp$ serving as a neutral element such that $\perp \oplus \perp = \perp$ and $x \oplus \perp = x$ for each $x \in \mathcal{D}$.

Handling More Keys When multiple distinct keys are present in the stream, PLS produces a stream of pane results $\mathcal{P}_i^k$, with i the progressive identifier of the pane and k the key attribute. In such a scenario, WLS keeps a B-PAT for each existing key.

Count-Based Windows So far, we have always assumed time-based windows in defining PAT and B-PAT. The approach supports count-based windows as well, where w, s, and p are expressed in a number of tuples.

V. Implementation

In this section, we provide the details of the CUDA implementation of Springald.

A. PLS Implementation

PLS receives tuples in batches allocated in device memory by the preceding operator (see Section III-A). For each batch, PLS applies a set of transformation steps on the GPU. Some transformations have been directly implemented as CUDA kernels, others are device primitives provided by Thrust.[3] In the first step, the pane identifiers are extracted from the tuples of the batch in parallel through a first kernel called *lifting*. Let t be an tuple, its pane identifier is $\lfloor t.ts/p \rfloor$, where p is the pane length. Then, tuples are sorted in decreasing order of pane identifiers through a thrust :: sort_by_key primitive. Finally, tuples having the same pane identifier are aggregated through a thrust :: reduce_by_key using the binary operator $\oplus$ provided by the user.

PLS keeps the set of pane results that are not complete. They are stored in a circular buffer pending, where num_pending is the number of stored pane results, while size is the capacity. The variable tail points to the position of the oldest element in the buffer (if not empty), while first_id is the identifier of the oldest non-complete pane (it might be or not present depending whether num_pending>0). PLS launches a kernel that aggregates the partial pane results in the input batch after the thrust :: reduce_by_key with the existing results having the same identifiers stored in the buffer pending. Such a kernel is shown in Fig. 7. Since we work with uni-dimensional arrays, we arrange CUDA threads in a one dimensional grid of one-dimensional blocks.

In the code, new_panes is the batch of pane results from the previous steps (i.e., after the thrust :: reduce_by_key), while num_new is its size. Threads of the kernel are assigned to the results in new_panes using a grid-stride *for* loop. At line 10, each thread computes for the pane result in new_panes at position i its corresponding position in pending. If this position already contains a partial aggregate for that pane (lines 11-13), it will

[3]Thrust is a C++ template library for CUDA based on the Standard Template Library, https://github.com/NVIDIA/thrust

```
 1: template<typename Agg_t, typename Op_t>
 2: __global__ void Aggregation(Op_t op, Agg_t *new_panes,
 3:                             size_t num_new, Agg_t *pending,
 4:                             size_t num_pending, size_t tail,
 5:                             size_t first_id)
 6: {
 7:   int id = threadIdx.x + blockIdx.x * blockDim.x;
 8:   int num_threads = gridDim.x * blockDim.x;
 9:   for (size_t i=id; i<num_new; i+=num_threads) {
10:       size_t pos = (tail + new_panes[i].id - first_id) % size;
11:       if (num_pending > 0) && // the pane exists
             (new_panes[i].id < first_id + num_pending) {
12:           op(pending[pos], new_panes[i], pending[pos]);
13:       }
14:       else { // the pane does not exist
15:           panes[pos] = new_panes[i];
16:           size_t num = 0;
17:           if (i == num_new-1)
18:               num = new_panes[i].id - (first_id + num_pending);
19:           else
20:               num = new_panes[i].id - new_panes[i+1].id -1;
21:           size_t j=1;
22:           size_t ll = (pos + size - 1) % size;
23:           while (j <= num) { // creating all the missing panes
24:               if (new_panes[i].id - j >= (first_id + num_pending) {
25:                   new (&(pending[ll])) Agg_t();
26:                   ll = (ll + size - 1) % size;
27:               }
28:               j++;
29:           }
30:       }
31:   }
32: }
```

Fig. 7. CUDA kernel updating the pending pane results with the data in the received batch after a reduce_by_key.

be updated by calling $\oplus$ at line 12. Otherwise (lines 14-30), the aggregate is copied at that position of pending (line 15). At lines 17-20, the thread computes how many new pane results must be initialized. For example, if the aggregate at position i of new_panes has identifier 9, and first_id + num_pending is equal to 6, the thread initializes the aggregates for panes with identifiers $6, 7, 8$ in pending. This initialization is done at line 25, where the default constructor of Agg_t initializes the result to $\perp$.

The capacity of the buffer, indicated by size, must accommodate all new pane results to be added with the batch computation. This is done by checking the size before calling the kernel. In case pending is not large enough, a resizing-up method is called. The new size is the minimum between double of the current capacity and the number of slots required to store all pending results from first_id to the highest pane identifier seen in the batch up to now. In order to regulate memory utilization and mitigate excessive resource occupation, pending is resized down to half of its current capacity if num_pending/size is below $1/4$.

B. WLS Implementation

WLS keeps a B-PAT structure for each key to compute window results. We show the memory layout, and the CUDA kernels for building and updating the tree, and to compute all windows in parallel.

Memory Layout We store a size-m B-PAT as a flat array T of size $2m - 1$, whose elements have type $\mathcal{D}$. The first m positions store the leaves (level 0), and the next $m/2$ positions the internal nodes at level 1, up to level $\log(m)$ representing the last position of the array that stores the root. Fig. 8 shows this layout for the B-PAT in Fig. 6. With this design, the position of the parent of the node at position i is computed as `(i >> 1) | m`, where `>> 1` is the logical shift right of one bit, while | is the bitwise logical OR.

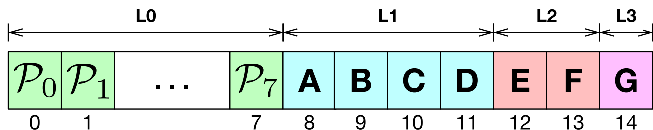

Fig. 8. Memory layout of the B-PAT in Fig. 6.

```
 1: template<typename Agg_t, typename Op_t>
 2: __global__ void Init_Level(Op_t op,
 3:                            Agg_t *level1,
 4:                            Agg_t *level2,
 5:                            size_t level2Size)
 6: {
 7:     int id = threadIdx.x + blockIdx.x * blockDim.x;
 8:     int num_threads = gridDim.x * blockDim.x;
 9:     for (size_t i=id; i<levelBSize; i+=num_threads) {
10:        op(level1[i*2], level1[i*2+1], level2[i]);
11:     }
12: }
```

Fig. 9. CUDA kernel initializing level2 by reading the elements of level1 of the B-PAT.

The size of T can be reduced by considering that we never use the internal nodes at levels higher than $\log(n)$, as already discussed (n is the number of panes per window). So, the actual size can be bounded to $\sum_{i=0}^{\lceil \log(n) \rceil} m/2^i$.

Building the B-PAT The initialization of the B-PAT starts when the first m pane results are available from PLS and written in the first m positions of T (if m is not a power of two, the remaining leaves are initialized to $\perp$). The initialization consists of launching a sequence of CUDA kernels to initialize all levels of the tree. Each kernel is responsible for initializing level k by reading the elements at level $k - 1$. So, kernels are sequentially launched. Fig. 9 shows the code of the kernel where level1 is the input level while level2 is the output level.

The code uses a binary operator op having signature Agg_t (const Agg_t&, const Agg_t&, Agg_t&), so aggregating the first two inputs and writing the result in the last input.

Updating the B-PAT When s_b new pane results are made available by PLS, they overwrite the positions of the array corresponding to the oldest pane results. Furthermore, all the internal nodes affected by such a change are updated. Again, the CUDA implementation consists of a sequence of kernel invocations one for each level. However, differently from the initialization kernel, only a subset of the elements at level $k - 1$ are considered by updating a fraction of the element at level k (i.e., only the ones affected by the changes). Fig. 10 shows the code of the kernel.

In the code, the input sizeUpdate is the number of elements of level2 to be updated, while offset is the position of the first element to be updated. Since the internal nodes to update can be contiguous in memory, or they can wrap around, we use the modulo operation at line 12.

Computing Windows Fig. 11 shows the kernel to compute all the windows stored in the B-PAT. The kernel takes in input the number of leaves numLeaves of the tree and the value m corresponding to the number of panes composing n_w consecutive windows. As said, the two values differ if m is not a power

```
1: template<typename Agg_t, typename Op_t>
2: __global__ void Update_Tree(Op_t op,
3:                             Agg_t *level1,
4:                             Agg_t *level2,
5:                             size_t offset,
6:                             size_t level2Size,
7:                             int sizeUpdate)
8: {
9:     int id = threadIdx.x + blockIdx.x * blockDim.x;
10:    int num_threads = gridDim.x * blockDim.x;
11:    for (size_t i=id; i<sizeUpdate; i+=num_threads) {
12:        size_t idx = (i + offset) % level2Size;
13:        op(level1[idx*2], level1[idx*2+1], level2[idx]);
14:    }
15: }
```

Fig. 10. CUDA kernel updating the internal nodes of level2 after the modification of the oldest s_b leaves.

```
1: template<typename Agg_t, typename Op_t>
2: __global__ void Compute(Op_t op, Agg_t *T,
3:                         Agg_t *results, size_t offset,
4:                         int numLeaves, int m,  int n,
5:                         int sp, int Nw)
6: {
7:     int id = threadIdx.x + blockIdx.x * blockDim.x;
8:     int num_threads = gridDim.x * blockDim.x;
9:     for (size_t i=id; i<Nw; i+=num_threads) {
10:        int wS = (offset + i * sp) % m;
11:        int panes = n;
12:        while(panes > 0) {
13:            wS = wS >= m ? 0 : wS;
14:            int  range = wS == 0 ? m : ( wS & -wS );
15:            int pow = nextPowerOf2(panes);
16:            range = range < pow ? range : pow;
17:            int tr = range;
18:            int tn = wS;
19:            while (tr > 1) {
20:                tn = (tn >> 1) | numLeaves;
21:                tr >>= 1;
22:            }
23:            op(results[i], T[tn], results[i]);
24:            int oldwS = wS;
25:            wS += range;
26:            range = wS >= m ? m - oldwS : range;
27:            panes -= range;
28:        }
29:    }
30: }
```

Fig. 11. CUDA kernel to compute the windows in the B-PAT.

of two, and numLeaves is the smallest power of two greater or equal to m.

Each CUDA thread, assigned to the window starting with the leaf at position wS, executes a *while* loop until there are pane results to aggregate (variable panes). Initially panes is equal to the number of panes per window $n = w/p$. Then, it finds how many panes can be aggregated by looking at the ancestors of the leaf at position wS. This is done by leveraging some bitwise operations. The operation `wS & -wS` (line 14) returns the greatest power of two that is a divisor of wS. Let wS be 6, `6 & -6 = 110 & 010 = 010 = 2`. This number represents the largest sequence of panes starting from the one at position wS that can be aggregated by an ancestor of the leaf at position wS. By referring to Fig. 6, node D can be used as it stores the aggregate of $\mathcal{P}_6$ and $\mathcal{P}_7$, while earlier ancestors store aggregates computed considering other leaves preceding the one at position wS. For all leaves having an odd position, this number is always 1 so the aggregate in the leaf is used directly. The leaf at position 0 is a special case. Since `wS & -wS` would return zero, we force this case (line 14) to return m (8 in the example), because the aggregate of all the leaves is stored in the root. This number minus 1 also indicates how many levels we have to climb to find this internal node (*while* loop at lines 19-22). Once found the position of that node (tn), we use its stored aggregate to update the window result (line 23). Then, we update panes and we repeat the loop until panes becomes zero.

C. Handling Multiple Keys

When more keys exist, WLS keeps an internal B-PAT for each key. These structures are allocated in the device memory once the first pane result associated with a new key reaches completion, as determined by a watermark. Handling multiple keys mainly affects the behavior of PLS, since separated buffers of pending pane results are kept one per key. Once a new batch has been received, a preliminary additional step is applied to extract the key attributes (through a CUDA kernel), and the sort$_b$y$_k$ey primitive described in Section V-A sorts the tuples in order to group all tuples having the same key attribute in contiguous positions (they are internally sorted in decreasing order of pane identifier, as already discussed). Then, the same steps are repeated for each portion of the batch having the same key, in order to update different B-PAT structures by the WLS.

VI. Evaluation

In this section, we first focus on the impact of different parameters such as the number of CPU threads involved in the processing, and the batch size, and we evaluate different configurations in terms of window length and slide. Next, we consider the impact of the aggregation function. Then, we study the performance resilience of SpringALD with different levels of stream disorder. Finally, an experimental comparison against state-of-the-art competitors is given.

A. Experimental Setup

Hardware Platform We conduct all the experiments on a machine running Ubuntu `22.04` with kernel `5.15.0-82`. It has two CPUs AMD EPYC 7551 with 128 GiB of RAM and 32 cores working at 2 GHz each. Each core has an L1D of 32 KiB, an L2(I+D) of 512 KiB, and each group of four cores shares an L3(I+D) of 8 MiB (for a total of 64 MiB). The machine mounts a PCIe board with an NVIDIA Ampere30 GPU having 56 Stream Multi-processors, each with 64 FP32 cores and 32 FP64 cores, for a total of 5,376 FP32+FP64 cores. The amount of GPU memory is 24 GiB of HBM2. The GPU works at 930 MHz with a peak of 1,440 MHz of boost clock speed. In all our experiments, hyper-threading has been disabled and we never use more threads than CPU cores available.

Benchmarks We set up a synthetic experiment with a parametric number of sources, which generate tuples at their maximum capacity, one instance of SpringALD, and a sink operator. The sources generate synthetic data streams of 32-byte tuples, each consisting of four 32-bit integer numbers, two single-precision floating-point numbers, and a 64-bit timestamp. Integer and

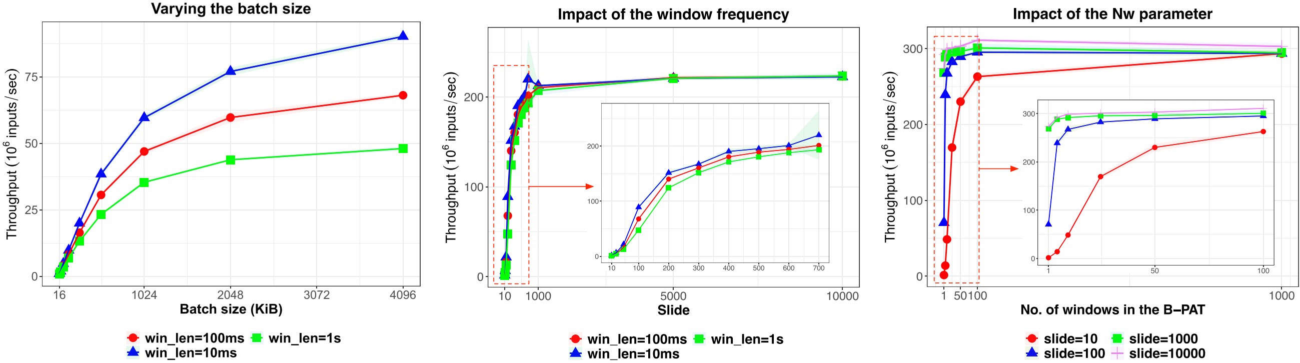

Fig. 12. Impact of the chosen batch size (*left*). Impact of the slide parameter of time-based windows adopted by SPRINGALD (*center*). Impact of the n_w parameter in the achieved throughput by SPRINGALD (*right*).

TABLE I
PARAMETERS CONSIDERED IN THE EXPERIMENTS

Symbol	Description
w, s	Window length and slide in time units or in no. of tuples
b	Batch size in no. of tuples. Batches are produced by the sources and transmitted to SPRINGALD
n_w	Number of consecutive windows represented in the B-PAT
f_a	Aggregation function (associative and commutative)
k	Number of distinct values of the key attribute
t	Parallelism (i.e., number of threads) executing the sources

floating-point numbers are drawn from a uniform distribution in the interval $(0, 1000)$. One of the integers is interpreted as the key attribute. Results produced by SPRINGALD are transferred back to the host memory and read by a Sink operator. Each experiment is run 10 times, and we report the 95th confidence interval (shaded area in the plots). In addition to such synthetic experiment, in the comparison with existing SPEs, we consider the Yahoo Streaming Benchmark [27] (YSB) as a realistic application. Such a benchmark emulates an advertisement application consisting of a source, a filter, a map (joining with a static table), a windowed aggregration and a Sink. It is configured with $w = 1$ sec, $s = 10^4$ usec and 1,000 dinstinct keys drawn from a uniform distribution.

Baselines We consider two existing frameworks that have significant relationships with our work (justifications for this choice will be given later in the text). The first is SABER [20], a system targeting GPUs for accelerating sliding-window aggregates over data streams. This system has been developed in Java using OpenCL for GPU exploitation. It supports hybrid CPU+GPU processing by computing the stream in batches. For each batch, it computes partial aggregates for different window fragments in parallel on CPU or GPU cores, which are aggregated into window-wise results by a sequential merge phase. Differently from SPRINGALD, SABER assumes the input stream rigorously ordered by timestamps. The second baseline is SCOTTY [22], an aggregation framework designed for out-of-order streams that does not support GPU devices. Section VII will give further details about these two systems.

Compilation Toolchain We use `gcc` version `12.3.0`, and `nvcc` version `12.2`. The machine uses the CUDA driver version `535.104.05`. For the baselines, SABER uses Oracle JVM 8, while SCOTTY runs on Flink `1.14.4`.

Studied Parameters Table I lists the main configuration parameters and a short description of each of them.

B. Performance Analysis

The performance of SPRINGALD is affected by different parameters. The first is the size of batches received by the SPRINGALD operator in the data-flow graph. Fig. 12 (left) shows the results of the first experiment conducted using f_a˜ = ˜sum over highly-frequent time-based windows sliding every 100 usec. We study three window lengths equal to 1 sec, 100 ms, and 10 ms, and we report the peak throughput measured with different batch sizes ranging from 16 KiB to 4 MiB. In this set of experiments, we set $n_w = 1$. Furthermore, we generate a stream without keys.

The results confirm that higher throughput is achieved with smaller window lengths since fewer panes need to be aggregated to compute window results. On average, the throughput with windows of 10 ms is 17% higher and 33% higher than the one measured with windows of 100 ms and 1 second respectively. Increasing the batch size has a remarkable impact since host-device transferring overheads can be amortized. Increasing the batch size from 16 KiB to 4 MiB yields to a throughput improvement of 93×, 77×, and 62× with windows of 10 ms, 100 ms, and 1 second respectively. Batch sizes larger than 4 MiB do not produce significant improvements since throughput reaches a plateau near 90×10^6, 68×10^6 and 48×10^6 inputs/sec with the three window lengths respectively.

> *Observation 1.* SPRINGALD needs proper batching of the data stream to achieve high throughput and amortizing data transfer costs. Bathes of a few megabytes are sufficient for this purpose.

Fig. 12 (center) shows the results of the second experiment where we fix the batch size to 4 MiB and we change the slide parameter from 10^1 to 10^5 usec. The experiments consider

the same window lengths of 10 ms, 100 ms, and 1 second, and still $n_w = 1$ windows in the B-PAT. With slides that are greater than 1000, SPRINGALD achieves approximately the same throughput for all the considered window lengths (nearly 220 million inputs/second). With small slides (i.e., more frequent windows), the throughput is slightly lower with longer windows, as expected. With a slide smaller than 1000 usec (shown in the zoomed subplot in the same figure), throughput is 12% and 21% lower with windows of 100 ms and 1 second compared with smaller windows of 10 ms respectively.

Using $n_w > 1$ has the advantage of increasing the amount of data parallelism to effectively exploit the GPU device (i.e., B-PAT has more leaves and internal nodes). Furthermore, since the B-PAT leaves are updated every time new $s_b = (n_w \cdot s)/p$ complete pane results are made available by PLS, with higher values of n_w we can amortize the transferring cost between PLS and WLS, and the overheads of the kernel in Fig. 10.

Fig. 12 (right) shows the results by varying the parameter n_w from 1 to 1000. We measure the throughput with a window length of 1 second and four different slides of 10^1, 10^2, 10^3, and 10^4 usec. We fix the batch size to 4 MiB, and we use $f_a = \mathsf{sum}$ without keys. As we can observe from the figure, greater values of n_w are useful to optimize throughput. This is evident mainly with small slides. With a slide of 10^1 usec, the initial throughput with $n_w = 1$ is extremely limited: only 1.5 million inputs/sec, which increases up to 293 million inputs/second with $n_w = 1000$. The improvement is remarkable (197×), while it is still evident but smaller with greater slide values: $4.3\times$ with a slide of 100 usec, and of 15% with slides of 10^3 and 10^4 usec. This confirms that increasing the number of consecutive windows represented in the B-PAT is pivotal.

> *Observation 2.* SPRINGALD provides high throughput with any configuration of (w, s) parameters provided that a properly sized n_w value is chosen.

The last experiment of this part focuses on the impact of the parameter t in Table I, i.e., the number of sources connected to SPRINGALD (i.e., each running in a dedicated CPU thread). More threads are of great importance to provide new batches to compute, avoiding the generation phase of the stream being a bottleneck. Fig. 13 shows the results by varying the threads from 1 to 64 (filling all CPU cores of the machine). Results have been collected with different batch sizes and fixing the window length and slide to 1 second and 10^4 respectively. Data streams in this experiment are still provided without distinct keys, using $f_a = \mathsf{sum}$, and n_w is set to 100 to achieve the highest throughput possible, as previously discussed.

As we can observe, a proper combination of batch size and parallelism allows SPRINGALD to achieve high throughput, up to 300 million inputs/second. With a small batch size of 256 KiB, scalability is limited and the highest throughput is 50 million inputs/second, using four threads generating data by partitioning the input dataset. With a batch size of 4 MiB (already used in the previous experiments as the default size), the highest throughput is measured with 20 threads (showing a scalability of about 14×). It should be observed that increasing the number of threads beyond this limit does not pay off: the throughput provided by SPRINGALD is limited by the PCIe bandwidth.

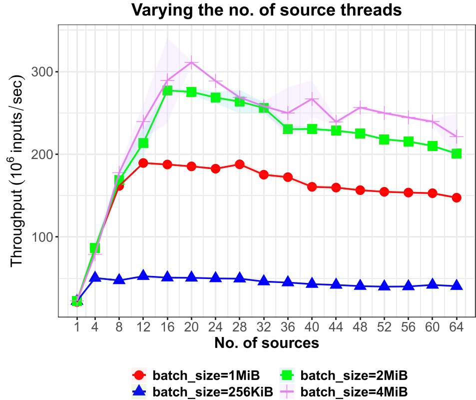

Fig. 13. Throughput provided by SPRINGALD by changing the number of threads and using different batch sizes.

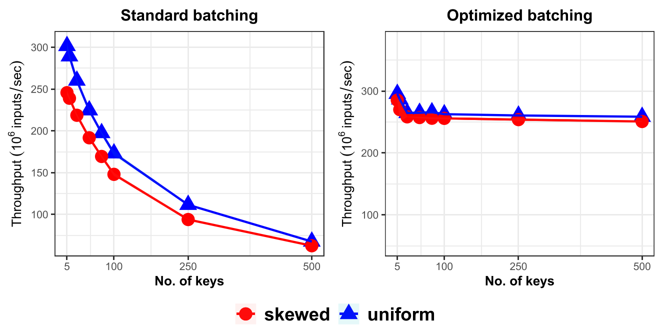

Fig. 14. Impact of the number of keys and their distribution.

> *Observation 3.* Batches need to be continuously provided to SPRINGALD by parallel sources in the data-flow graph. Parallelization of the input stream generation/reading (e.g., from files) is pivotal.

C. Impact of Keys

In this part, we analyze the impact of having more keys in the stream.

Fig. 14 (left) shows the results conducted using $f_a = \mathsf{sum}$, a window of length 1 second, and a slide of 10^4 usec. We vary the number of distinct keys in the stream, from 1 to 500, and we consider two distributions: uniform and a Zipfan distribution with skewness parameter 0.9, which generates a highly skewed distribution of keys in the stream. So, we can study the impact of the number of keys and of their imbalance. The throughput with more keys drops significantly. In the experiment with 500 distinct keys, throughput decreases by $4.54\times$ (uniform) and $4.11\times$ (skewed) compared with the execution without distinct keys. Throughput is lower with the skewed distribution compared with the same execution with the uniform distribution of keys.

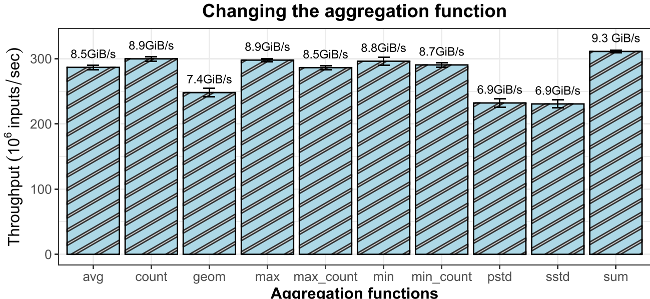

Fig. 15. Peak throughput (in inputs/sec and in GiB/s) by SPRINGALD using different aggregation functions.

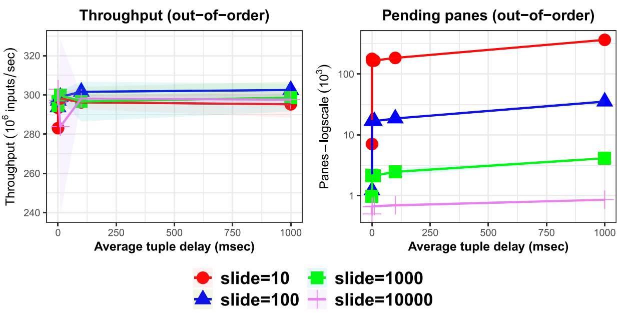

Fig. 16. Performance of SPRINGALD with out-of-order data streams.

Throughput can be improved with a proper batching strategy performed by the Emitter nodes (see Section III-A), which are the entities in charge of preparing batches transmitted to SPRINGALD from the replicas of the preceding operator in the data-flow graph. Instead of filling the batches in FIFO order, as done in the previous experiment, we have designed a more optimized approach where batches are prepared in order to not contain more than $\delta_k > 0$ different keys, where δ_k is a user-defined parameter. Fig. 14 (right) shows the results by repeating the previous experiment with the optimized Emitters working with $\delta_k = 25$. Throughput decreases smoothly with more keys and then reaches a plateau with several hundreds of keys. Throughput remains unaffected because optimized Emitters keeps more pending batches and emit them once they have been completely filled (however, they will have no more than δ_k distinct keys each). The performance drop is now moderate: from 1 to 500 keys we lose -16% (uniform) and -17% (skewed).

The reason for the good performance of SPRINGALD with skewed distributions and optimized batching is that tuples with the same key are processed in parallel on the device owing to the associativity of the binary window operator to compute. Even if most of the tuples belonged to the same subset of keys, input batches would be processed in parallel on the device. The key distribution has instead effect on latency, since in case of a skewed distribution more time is needed to buffer the required number of tuples of those keys with a relatively low probability, by delaying their computation until a sufficient threshold is achieved.

> *Observation 4.* SPRINGALD tolerates streams with hundreds of different keys and skewed distributions with small impact on the provided throughput.

D. Changing the Aggregation Function

The aggregation function plays a minor role in the achieved throughput, although we have measured some differences. Fig. 15 shows the peak throughput achieved by SPRINGALD with different functions. We report over each bar the exploited bandwidth in GiB/s. While sum provides the best result (very similar to count, max and min), others achieve slightly lower throughput (e.g., avg, max_count and min_count). Lower results are achieved with sstd and pstd that require more calculations per input to update each partial aggregate.

E. Impact of Out-of-Order Data Streams

SPRINGALD is the first GPU-accelerated sliding-window operator working with out-of-order streams. Therefore, in this part we evaluate its effectiveness in coping with disordered streams, with a controlled average delay of tuples produced by the sources. Results are reported in Fig. 16 (left), where we use $f_a = \text{sum}$, a window of 1 second, different slides of 10^1, 10^2, 10^3 and 10^4 usec, and a fixed batch size of 4 MiB. We vary the average delay of tuples from zero (in order) to a large delay of 1 second. Delays are uniformly distributed.

As we can see in Fig. 16 (left), SPRINGALD processes out-of-order data streams as fast as in-order streams. Indeed, throughput remains constant by increasing the average input delay, since SPRINGALD updates pane results efficiently by processing tuples within each batch in an out-of-order fashion. The consequence of the stream delay is instead evident in the memory consumption of SPRINGALD, which is represented in Fig. 16 (right). We measure the average number of pending pane results kept by the PLS, which increases proportionally to the average delay since more pending results need to be kept and updated. Then, when an input watermark closes a set of pane results, they are copied in the corresponding B-PAT by the WLS. This experiment confirms the effectiveness of SPRINGALD in processing out-of-order streams efficiently.

> *Observation 5.* SPRINGALD is insensitive to the degree of out-of-orderliness of the stream.

F. Speedup

We study the speedup of using the GPU compared with a handmade C++ parallelization of the PLS and WLS stages running in parallel on the CPU cores. Fig. 17 shows the results of two experiments by variating two important parameters. The first (left) shows the speedup by varying the slide parameter, while the second (right) variates the number of distinct keys. The CPU baseline and SPRINGALD are configured with the best parameters in terms of batch size and parallelism degree. We use $f_a = \text{sum}$, $n_w = 100$ (for SPRINGALD), and a window of 1 second.

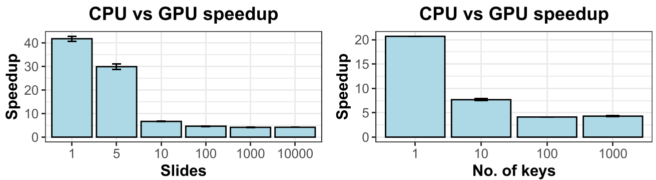

Fig. 17. GPU versus CPU speedup by varying the slide parameter (*left*) and the number of keys (*right*).

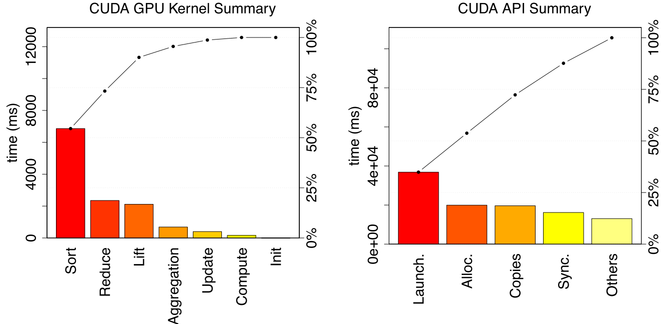

Fig. 18. Profiling results with NVIDIA Nsight systems.

Fig. 17 (left) shows the results with different slides while the number of keys is fixed to 100. With smaller slides, so a more computationally-demanding scenario with very frequent windows, the speedup in favor of GPU processing is generally better. It ranges from $40\times$ to $4.1\times$ with the considered slide values. Fig. 17 (right) shows the speedup by changing the number of keys. The slide parameter has been fixed to 10^4 usec. The highest speedup of $20.6\times$ is obtained with one key, and decreases with more keys with a minimum of $4.1\times$ since the number of keys expresses the parallelism exploitable by the CPU baseline.

G. Profiling Results

We study the execution time breakdown of a short run of SPRINGALD with windows of 1 second, slide of 10^4 usec, one key, $f_a = \textsf{sum}$, $n_w = 100$, and the best batch size to achieve the highest speedup. This analysis has been conducted by profiling the execution with the Nsight Systems tool (nsys) provided by NVIDIA. Fig. 18 shows two analyses corresponding to two options provided to the profiler: $\textsf{–reportcuda}_\textsf{g}\textsf{pu}_\textsf{k}\textsf{ern}_\textsf{s}\textsf{um}$ and $\textsf{–reportcuda}_\textsf{a}\textsf{pi}_\textsf{s}\textsf{um}$. The first profiles the kernel execution times, while the second the ones of the main CUDA API calls.

As shown in Fig. 18 (left), the kernels processing batches (i.e., the ones used by the $\textsf{thrust :: sort}_\textsf{b}\textsf{y}_\textsf{k}\textsf{ey}$ and $\textsf{thrust :: reduce}_\textsf{b}\textsf{y}_\textsf{k}\textsf{ey}$ primitives, and the lifting kernel) are executed more frequently and take a higher portion of the execution time. As expected, the other kernels are executed less frequently since they are launched to aggregate pane results (once they are closed by a watermark) into window results. The contribution of kernels used to build or update the B-PAT is instead negligible. So, even if their parallelism exponentially decreases by raising the levels of the tree, their impact is amortized with the chosen n_w value.

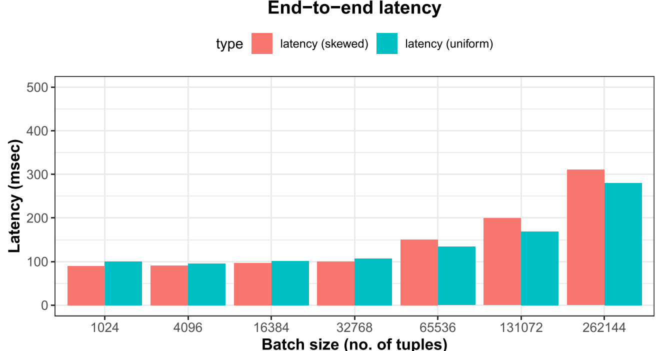

Fig. 19. End-to-end average latency evaluation.

Fig. 18 (right) shows the execution times of the main CUDA API calls. The component with the highest contribution are the API to launch kernels, while H2D and D2H memory copies (and synchronization to wait for kernels completion) are below 20% both, reflecting a good efficiency of SPRINGALD. Furthermore, after an initial transient phase where the SPRINGALD implementation allocate GPU memory for each new key in the data stream, the overall device utilization becomes stable and it is within the range 90-96%.

H. Latency

Fig. 19 shows the results of the end-to-end latency (i.e., source-to-sink elapsed time). We consider windows of 1 second, slide of 10^4 usec, 1,000 keys, $f_a = \textsf{sum}$, $n_w = 100$. In this experiment, we change the batch size and the key distribution. Latency increases with greater batches since the buffering delay dominates. It is interesting that with a skewed distribution of keys, latencies become higher with large batches. This is due to the contribution of keys having a low probability, which spend a large time in the *emitter* nodes responsible for properly buffering them in sufficiently large batches. However, SPRINGALD always provides bounded latencies in the order of a few hundred of milliseconds, which is comparable with the latency of traditional scale-out SPEs like Flink.

I. Comparison With State-of-the-Art

We compare SPRINGALD with two state-of-the-art (SOA) competitors. For the first competitor, our goal was to select a prototype allowing window aggregation on heterogeneous architectures comprising CPU+GPU devices. Our choice has fallen on SABER [20] which, although limited to in-order streams only, allows leveraging a GPU device to accelerate the aggregation process. Other prototypes described in research papers (such as the ones in [21]) did not provide a public repository or binaries. About the second competitor, we looked for a prototype coping with out-of-order streams. Since no GPU alternative is available for such a scenario, we consider SCOTTY [22], an out-of-order CPU-only window aggregator made available in different SPEs (i.e., Flink, Storm, and Spark). Other CPU-only options like the ones in [18], [19] still have in-order stream processing constraints.

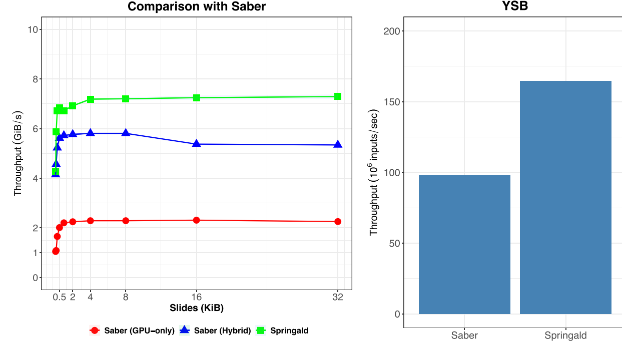

Fig. 20. Comparison with SABER.

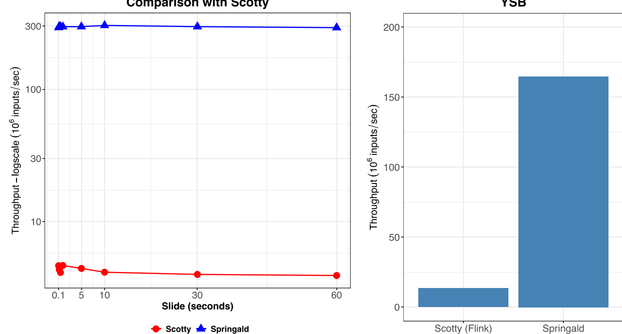

Fig. 21. Comparison with SCOTTY.

1) Comparison With SABER: Fig. 20 (left) shows the result of the same experiment proposed in the paper of Koliousis et al. [20] with $f_a = \mathsf{avg}$, count-based windows of 32 KiB (equal to 1024 tuples of 32 bytes each) and a slide ranging from 64 bytes to 32 KiB. The general qualitative behavior for all SPEs is that with larger slides throughput increases. For SABER, the figure shows two independent lines: the first (GPU-only) is the execution where only the GPU (and one host thread for launching kernels and performing host-device data transfers) is employed in the sliding-window execution; the second (*hybrid*) uses the full potential of SABER, which processes sliding windows using both CPU cores and the GPU.

As we can observe, the performance of SABER is mainly due to the CPU contribution, since the GPU-only version achieves a peak throughput of 2.3 GiB/s at most, while the hybrid version reaches 5.34 GiB/s with large slides. SPRINGALD outperforms both versions: it achieves a peak throughput of 7.30 GiB/s, and it exhibits +260% and +24% higher throughput on average compared with SABER (GPU-only and hybrid versions respectively) with the different considered slide values. We further point out that the superior performance exhibited by SPRINGALD comes with the additional feature of processing out-of-order data streams efficiently, which is not supported by SABER. As reported in Fig. 20 (right), the superior performance of SPRINGALD is confirmed also with a real-world application like the YSB, where the throughput improvement is of 68% in the best conditions of both systems.

2) Comparison With SCOTTY: Fig. 21 (left) shows a comparison between SPRINGALD and SCOTTY. The latter is configured to be used in Apache Flink. The experiment considers a time-based window of 60 seconds and different sliding values ranging from 100 ms to 60 seconds. We use an average delay of 1 second to emulate out-of-order streams that are supported by both SCOTTY and SPRINGALD.

Although SCOTTY has several analogies with our design (it is based on a *slicer* component similar to our PLS), it is implemented in Java and for CPU processing only. Furthermore, it leverages parallelism only by processing tuples of different keys by different threads, while the ones with the same key are processed sequentially. This leads to a significant improvement in favor of SPRINGALD. The difference is of two orders of magnitude: SCOTTY processes at most 4 million inputs/second, while SPRINGALD more than 300 million, exhibiting a stable behavior with all the considered slides. The reasons for this improvement are related to the GPU exploitation done by SPRINGALD. Furthermore, SCOTTY inherits the overheads of Flink as studied by Zhang et al. [28]. However, SCOTTY provides some features not currently supported by SPRINGALD, such as running aggregates using different window definitions simultaneously by sharing partial results, a feature that we would like to add to SPRINGALD in the future. Such a superior performance is shown also in the YSB, with a significant improvement in throughput of 12.2×.

Observation 6. SPRINGALD revealed more efficient than GPU-based SOA solutions for in-order stream processing. Furthermore, its throughput is remarkably higher than CPU-based out-of-order streaming aggregators integrated into popular open-source SPEs.

VII. RELATED WORKS

Algorithms for Sliding-Window Aggregation Sliding-window aggregation has been the subject of extensive research over the years. The bucket algorithm [9] does not leverage partial overlapping between consecutive windows, which are computed from scratch. Techniques to share computations have been proposed in the literature [10], [11], [29], [30], where partial aggregates are stored in different data structures (e.g., binary trees [12], stacks [15], deques [13]). The idea of sharing partial results has been extended to the multi-query scenario [14], where more queries apply different aggregates over sliding windows over the same stream. Most of the previous approaches support in-order streams (i.e., ordered by timestamps). Out-of-order data streams pose additional challenges to sliding-window aggregation since a window is considered complete only when the first watermark closing it has been received. Techniques to deal with such a scenario have been presented in [16], [31]. Scotty [22] is a recent out-of-order operator supporting different window semantics and aggregation functions in popular SPEs like Flink, Storm, and Spark.

Type of Aggregation Functions Most of the research contributions provide support for associative and commutative aggregation functions, which represent the most common utilization of aggregates. Other kinds of aggregation functions exist, and they often pose additional challenges. FlatFat [12]

supports non-FIFO windows with aggregation functions that can be non-invertible and non-commutative. DIBA [15], providing worst-case constant aggregation time complexity, requires the aggregation function to be associative and the window semantics to be FIFO. Associative but non-commutative aggregations are also supported by SlickDeque [14], which however provides different implementations for invertible and non-invertible functions. Non-algebraic aggregates like holistic ones (e.g., count-distinct, mode, and quantiles) pose serious challenges to be computed incrementally since they need to maintain a global state for the entire aggregation. An approach for a subset of them has been also proposed [32].

Window-Based Semantics Count and time windows (both tumbling and sliding), which are the ones considered in this paper, adopt the FIFO semantics, i.e., the oldest input added to the window is also the first one that will be evicted. Non-FIFO semantics have been defined for specific use cases, such as delta-based windows [33]. Windows can also be classified into *forward context-free* (FCF) or *forward context-aware* (FCA) [23]. In FCF windows, the windows to which a new input belongs can be determined at the instant when that input arrives at the system. In FCA windows, this mapping cannot be established at that time, since it depends on inputs that will arrive in the future. Aggregates over FCA windows pose additional challenges to be parallelized [34].

Hardware Accelerated Sliding-Window Aggregation The last years have witnessed the rapid diffusion of SPEs accelerating sliding-window computations. StreamBox [17] is an SPE targeting multicores, where relational operators are scheduled onto a pool of workers for processing batches of data. The scheduling is performed in a centralized fashion, using lock-based primitives. Saber [20] is a relational SPE with support for both count and temporal windows. It is considered highly efficient, owing to its innovative support for hybrid architectures (it leverages both CPU and GPU). Its recent extension, which however supports only multi-core CPUs, is LightSaber [18]. Both Saber and LightSaber processes streams that are produced in-order by data sources. Grizzly [19] is a code generation approach to generate efficient code for relational stream processing. It targets sliding-window computations and does not provide support for out-of-order data streams. A recent SPE targeting integrated GPUs has been proposed [21]. Although it provides better performance than SABER on integrated GPUs, a comparison with it was not possible since the source code has not been publicly released.

VIII. Conclusions and Future Work

SPRINGALD is an operator processing count- and time-based sliding windows on GPUs. It tolerates out-of-order data streams in an effective manner. The experimental evaluation showcases the effectiveness of SPRINGALD, which provides a novel contribution since it accelerates sliding-window aggregates on GPU devices while tolerating out-of-order data streams. In the future, we plan to study the impact of GPU contention in scenarios where multiple SPRINGALD instances are used in the same application or in different applications running concurrently on the same machine. Furthermore, SPRINGALD can be extended to support multiple window definitions (i.e., with different length and slide parameters) applied simultaneously over the same stream.

References

[1] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge, U.K.: Cambridge Univ. Press, 2014.

[2] A. Arasu, S. Babu, and J. Widom, “The CQL continuous query language: Semantic foundations and query execution,” *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006. [Online]. Available: https://doi.org/10.1007/s00778--004-0147-z

[3] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl, “Algorithms for windowed aggregations and joins on distributed stream processing systems,” *Datenbank-Spektrum*, vol. 22, no. 2, pp. 99–107, 2022. [Online]. Available: https://doi.org/10.1007/s13222--022-00417-y

[4] D. Abadi et al., “Aurora: A data stream management system,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2003, Art. no. 666. [Online]. Available: https://doi.org/10.1145/872757.872855

[5] Y. Ahmad et al., “Distributed operation in the borealis stream processing engine,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2005, pp. 882–884. [Online]. Available: https://doi.org/10.1145/1066157.1066274

[6] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, “Out-of-order processing: A new architecture for high-performance stream systems,” *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 274–288, Aug. 2008. [Online]. Available: https://doi.org/10.14778/1453856.1453890

[7] A. Storm, “Apache storm,” 2023. [Online]. Available: https://storm.apache.org/

[8] A. Flink, “Apache Flink: Stateful computations over data streams,” 2023. [Online]. Available: https://flink.apache.org/

[9] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, “Semantics and evaluation techniques for window aggregates in data streams,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2005, pp. 311–322. [Online]. Available: https://doi.org/10.1145/1066157.1066193

[10] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, “No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams,” *SIGMOD Rec.*, vol. 34, no. 1, pp. 39–44, Mar. 2005. [Online]. Available: https://doi.org/10.1145/1058150.1058158

[11] S. Krishnamurthy, C. Wu, and M. Franklin, “On-the-fly sharing for streamed aggregation,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2006, pp. 623–634. [Online]. Available: https://doi.org/10.1145/1142473.1142543

[12] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, “General incremental sliding-window aggregation,” *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 702–713, Feb. 2015. [Online]. Available: https://doi.org/10.14778/2752939.2752940

[13] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis, “SlickDeque: High throughput and low latency incremental sliding-window aggregation,” in *Proc. 21st Int. Conf. Extending Database Technol.*, M. H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu, and K. Hose, Eds., 2018, pp. 397–408. [Online]. Available: https://doi.org/10.5441/002/edbt.2018.35

[14] G. Theodorakis, P. R. Pietzuch, and H. Pirk, “SlideSide: A fast incremental stream processing algorithm for multiple queries,” in *Proc. 23rd Int. Conf. Extending Database Technol.*, 2020, pp. 435–438. [Online]. Available: https://doi.org/10.5441/002/edbt.2020.51

[15] K. Tangwongsan, M. Hirzel, and S. Schneider, “Low-latency sliding-window aggregation in worst-case constant time,” in *Proc. 11th ACM Int. Conf. Distrib. Event-Based Syst.*, New York, NY, USA, 2017, pp. 66–77. [Online]. Available: https://doi.org/10.1145/3093742.3093925

[16] K. Tangwongsan, M. Hirzel, and S. Schneider, “Optimal and general out-of-order sliding-window aggregation,” *Proc. VLDB Endowment*, vol. 12, no. 10, pp. 1167–1180, Jun. 2019. [Online]. Available: https://doi.org/10.14778/3339490.3339499

[17] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, “StreamBox: Modern stream processing on a multicore machine,” in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 617–629.

[18] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk, “LightSaber: Efficient window aggregation on multi-core processors,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2020, pp. 2505–2521. [Online]. Available: https://doi.org/10.1145/3318464.3389753

[19] P. M. Grulich et al., "Grizzly: Efficient stream processing through adaptive query compilation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2020, pp. 2487–2503. [Online]. Available: https://doi.org/10.1145/3318464.3389739

[20] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "SABER: Window-based hybrid stream processing for heterogeneous architectures," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, 2016, pp. 555–569. [Online]. Available: https://doi.org/10.1145/2882903.2882906

[21] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, "FineStream: Fine-grained window-based stream processing on CPU-GPU integrated architectures," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2020, Art. no. 43.

[22] J. Traub et al., "Scotty: General and efficient open-source window aggregation for stream processing systems," *ACM Trans. Database Syst.*, vol. 46, 2021, Art. no. 1.

[23] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl, "Survey of window types for aggregation in stream processing systems," *VLDB J.*, vol. 32, pp. 985–1011, 2023. [Online]. Available: https://doi.org/10.1007/s00778--022-00778-6

[24] B. He et al., "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, Dec. 2009, Art. no. 21. [Online]. Available: https://doi.org/10.1145/1620585.1620588

[25] G. Mencagli, M. Torquati, A. Cardaci, A. Fais, L. Rinaldi, and M. Danelutto, "WindFlow: High-speed continuous stream processing with parallel building blocks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2748–2763, Nov. 2021.

[26] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High-Level and Efficient Streaming on Multicore*. Hoboken, NJ, USA: Wiley, 2017, pp. 261–280. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119332015.ch13

[27] S. Chintapalli et al., "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1789–1792.

[28] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze, "Revisiting the design of data stream processing systems on multi-core processors," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 659–670.

[29] G. Theodorakis, A. Koliousis, P. R. Pietzuch, and H. Pirk, "Hammer slide: Work- and CPU-efficient streaming window aggregation," in *Proc. Int. Workshop Accelerating Anal. Data Manage. Syst. Using Modern Processor Storage Architectures*, 2018, pp. 34–41.

[30] C. Zhang, R. Akbarinia, and F. Toumani, "Efficient incremental computation of aggregations over sliding windows," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, New York, NY, USA, 2021, pp. 2136–2144. [Online]. Available: https://doi.org/10.1145/3447548.3467360

[31] S. Bou, H. Kitagawa, and T. Amagasa, "CPiX: Real-time analytics over out-of-order data streams by incremental sliding-window aggregation," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 11, pp. 5239–5250, Nov. 2022.

[32] R. Wesley and F. Xu, "Incremental computation of common windowed holistic aggregates," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1221–1232, Aug. 2016. [Online]. Available: https://doi.org/10.14778/2994509.2994537

[33] B. Gedik, "Generic windowing support for extensible stream processing systems," *Softw. Pract. Experience*, vol. 44, no. 9, pp. 1105–1128, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2194

[34] G. Mencagli, M. Torquati, F. Lucattini, S. Cuomo, and M. Aldinucci, "Harnessing sliding-window execution semantics for parallel stream processing," *J. Parallel Distrib. Comput.*, vol. 116, pp. 74–88, 2018. [Online]. Available: https://doi.org/10.1016/j.jpdc.2017.10.021

Gabriele Mencagli is an associate professor with the Computer Science Department, University of Pisa, Italy. He is co-author of more than 90 peer-reviewed papers appeared in international conferences, workshops and journals, and of one book. His research interests include the area of parallel and distributed systems and data stream processing. He served in several Program Committees of international conferences and workshops, and he was the general chair of HPDC 2024. He is a member of the editorial board of the *Future Generation Computer Systems* (Elsevier) and *Cluster Computing* (Springer).

Patrizio Dazzi is a tenuered senior assistant professor with the University of Pisa. He deals with High-Performance Distributed Systems. He is the co-founder and co-leader of the Pervasive AI Laboratory. He is active in the community of large distributed systems, he has been an organizer of conferences and workshops, and promoted and edited journal special issues. He has co-authored more than 100 papers in international journals and conferences on topics related to High-Performance Distributed Systems.

Massimo Coppola is a researcher of Consiglio Nazionale delle Ricerche, member of the Institute for Information Science and Technologies (CNR-ISTI). His research interests include parallel and distributed platforms and programming frameworks, as well as their application to real-world use cases. His work encompasses large-scale heterogeneous parallel platforms, including cloud, edge computing, and continuum ones, as well as self-adapting mechanisms and data mining and machine learning applications. He is coauthor of more than 90 papers in peer-reviewed international conferences and journals, and he is currently associate editor of the *Frontiers in High Performance Computing*.

Open Access funding provided by 'Università di Pisa' within the CRUI CARE Agreement