Received February 26, 2019, accepted April 5, 2019, date of publication April 11, 2019, date of current version April 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2910312

GASSER: An Auto-Tunable System for General Sliding-Window Streaming Operators on GPUs

TIZIANO DE MATTEIS, GABRIELE MENCAGLI, DANIELE DE SENSI, MASSIMO TORQUATI, AND MARCO DANELUTTO

Department of Computer Science, University of Pisa, I-56127 Pisa, Italy

Corresponding author: Gabriele Mencagli (mencagli@di.unipi.it)

ABSTRACT Today's stream processing systems handle high-volume data streams in an efficient manner. To achieve this goal, they are designed to *scale out* on large clusters of commodity machines. However, despite the efficient use of distributed architectures, they lack support to co-processors like graphical processing units (GPUs) ready to accelerate data-parallel tasks. The main reason for this lack of integration is that GPU processing and the streaming paradigm have different processing models, with GPUs needing a bulk of data present at once while the streaming paradigm advocates a *tuple-at-a-time* processing model. This paper contributes to fill this gap by proposing `Gasser`, a system for offloading the execution of *sliding-window operators* on GPUs. The system focuses on completely general functions by targeting the parallel processing of *non-incremental queries* that are not supported by the few existing GPU-based streaming prototypes. Furthermore, `Gasser` provides an *auto-tuning approach* able to automatically find the optimal value of the configuration parameters (i.e., batch length and the degree of parallelism) needed to optimize throughput and latency with the given query and data stream. The experimental part assesses the performance efficiency of `Gasser` by comparing its peak throughput and latency against Apache `Flink`, a popular and scalable streaming system. Furthermore, we evaluate the penalty induced by supporting completely general queries against the performance achieved by the state-of-the-art solution specifically optimized for incremental queries. Finally, we show the speed and accuracy of the auto-tuning approach adopted by `Gasser`, which is able to self-configure the system by finding the right configuration parameters without manual tuning by the users.

INDEX TERMS Data stream processing, sliding-window queries, GPU processing, autotuning, self-configuring systems.

I. INTRODUCTION

Data streams have become commonplace in the realm of data science. An ever-growing number of mining tasks are applied to transient flows of data in a real-time manner rather than offline, on static, permanent datasets [1]. Examples are applications in finance, sensor networks, smart cities and many others.

With the proliferation of sensing devices and the overwhelming interest in 5G network technologies, input data rates have started to raise at a dramatic speed and are expected to skyrocket in the near future. The use of parallel processing models and architectures to speed up streaming tasks is of paramount importance to address the high-demanding performance requirements of time-critical applications. The design of Stream Processing Systems (SPSs) mostly targets *scale-out* scenarios using clusters of commodity machines [2]. Instead, the research of SPSs optimized for modern *scale-up* servers equipped with co-processors like GPUs (Graphical Processing Units) and FPGAs (Field Programmable Gate Arrays) is still in progress.

Relatively few papers have proposed prototypes of SPSs leveraging support to GPU processing [3], [4]. The common approach is to split the stream into disjoint batches, offload the processing of each batch on the GPU by exploiting data parallelism, and collect the results on the CPU. This idea has been applied to *sliding-window queries* [5] that repeat a user-defined function over the most recent *tuples* (input items)–the last $n > 0$ tuples or the ones received in the last n time units. One of the most relevant papers in this topic is `Saber` [6],

The associate editor coordinating the review of this manuscript and approving it for publication was Farhana Jabeen Jabeen.

where windows can span across several batches and may start or end in the middle of a batch. To express GPU parallelism in Saber, the supported queries must be decomposable by the user into two functions: the first producing a result per window fragment, the second assembling those results to build the result of the entire window. Examples of such queries are algebraic aggregates based on associative functions and sliding-window joins.

Gasser (*Gpu Accelerated Sliding-window StrEam pRocessing*) aims at filling two notable gaps within the literature to date. First, approaches like Saber cannot be adopted when the user-defined function is not decomposable, i.e. it is expressed by a non-incremental algorithm needing access to all the tuples of the window as a whole before starting the processing. Such case is often supported by SPSs (e.g., Apache Flink) to write streaming queries when incremental algorithms do not exist or, although available in the literature, they are hard to be implemented by a standard user. Since in those cases the computation is not executed incrementally, parallel processing techniques are a real challenge. Second, existing prototypes are hard to be tuned in terms of some system parameters like the batch size, which determines the efficiency of the parallel processing. Finding the right value may depend on the hardware and the stream characteristics that are often unknown before starting the processing.

This paper presents the motivation, the architecture and the run-time system of Gasser. The system has two main features that distinguish it from the state-of-the-art by addressing the two aforementioned gaps in the literature:

- it relies on the notion of *sliding batches* that allows the processing of consecutive windows to be offloaded on the GPU. Since the user-defined function is considered a blackbox by Gasser, to increase throughput our system executes distinct windows in parallel on the GPU cores. We call this model *inter-window parallelism*;
- to automatize its configuration with different queries and data streams, Gasser provides an *auto-tuning support* based on an online learning model to reach a system configuration (in terms of batch size and other setup parameters) achieving the optimal throughput.

By using Gasser the final users have two important advantages with respect to using traditional frameworks:

- they can exploit a GPU device to increase throughput for their streaming pipelines, feature not provided in traditional frameworks (e.g., Apache Flink and Storm). Furthermore, the use of the GPU is not limited to some predefined and built-in queries as in the few existing GPU-based research prototypes [6];
- the programmers are not involved in deciding when and how the GPU will be utilized: the system automatically decides which configuration is the best according to the workload conditions of the data stream and of the given query to execute.

In conclusion, Gasser *supports arbitrary windowed queries while exploiting the most effective level of parallelism and batching without manual tuning by the users*.

The next section of the paper provides an overview of the problem tackled by Gasser. Section III describes the Gasser architecture and the GPU kernel. Section IV presents the auto-tuning support. The evaluation is provided in Section V, while Section VI reviews the state-of-the-art. Section VII outlines the conclusion of our work and our future research directions.

II. PROBLEM OVERVIEW

In this section, we recall some basic concepts of streaming queries by focusing on the sliding-window paradigm. Then, we give the motivations of our work and we show the class of queries that we want to target in this research.

A. SLIDING-WINDOW QUERIES

Queries in the streaming domain are different from the ones in traditional databases. Streaming queries are standing, always on execution, and provide continuous sequences of results [5], [7]. In many applications the informational value of each tuple is time-decaying [8]. The approach is to maintain a *sliding window* of the most recent tuples whose content is updated as new tuples arrive while the oldest ones are periodically eliminated. Then, at every window activation the query logic goes over the tuples within the current window by producing a new output result.

Most of the SPSs provide support to sliding windows [9]. Different models have been introduced over the years. *Count-based windows* express both the *window length* $w > 0$—the past history captured by the window extent—and the *sliding factor* $s \leq w$—how much the window moves ahead when it slides—in number of tuples. When the window length and the sliding factor are expressed in time units (e.g., seconds, milliseconds), we are in presence of *time-based windows*. Besides these two basic definitions, other more exotic models have been proposed to support the users in specific application domains [5], [10]–[12].

B. WINDOW PROCESSING

The users develop streaming applications as dataflow graphs (topologies) of operators consuming input tuples and producing output results. Some of those operators use an interface to express sliding-window computations. SPSs provide different APIs, either supporting *incremental* or *non-incremental* processing.

When the query is provided with an incremental function, window results are incrementally computed by aggregating the tuples as they arrive. For commutative and associative functions, this allows for efficient computation because the system may avoid recomputing each window from scratch by reusing partial results of shared portions between consecutive windows. Optimized algorithms for incremental queries have been the subject of an intensive research for some notable queries (e.g., aggregates and quantiles [13]–[15]). This has

Listing

```
public abstract class ProcessWindowFunction<IN, OUT, KEY, W extends Window>
implements Function {
  public abstract void process(
          KEY key,
          Context context,
          Iterable<IN> elements,
          Collector<OUT> out) throws Exception;
  ...
}
```

Listing 1. Process window function abstract class.

also been studied for providing parallel solutions on GPUs (like in `Saber` [6]), where incremental queries allow the window processing to be split by computing partial results of different window fragments in parallel and then merging them to obtain final window results.

Most of the SPSs provide also a non-incremental API to instantiate operators with any offline algorithm that expects to receive all the window tuples before starting the processing. Since the input function is generic, this in general prevents to exploit results already obtained for preceding windows to save computation time. Listing 1 shows the non-incremental API provided in Apache `Flink` version `1.7.1`. The user implementing a non-incremental algorithm has to extend the `ProcessWindowFunction` abstract class by implementing the `process()` method with the logic to be executed for each triggered window. The method gets an `Iterable` containing all the elements of the window.

Non-incremental APIs improve the generality of SPSs by allowing the use of any already existing algorithm in streaming pipelines. This goes in the direction to make streaming frameworks more suitable for general data mining and machine learning tasks, which enhance the capability to extract actionable intelligence for decision makers. Indeed, limiting the support only for incremental or one-pass algorithms, although they are closer to the streaming context, cuts off the use of standard algorithms for traditional problems. Examples are classification [16] (e.g., decision trees, random forest, support vector machines, K-NN), and most of the inductive machine learning algorithms, which, in their traditional definition, need the training set available as a whole and do not apply incremental processing for building the model [17].

Unfortunately, GPU-based solutions for queries expressed with non-incremental functions on data streams are generally not available. `Gasser` fills this gap that we claim has practical relevance for the users. We will introduce a parallel approach that assumes the query function to be a *blackbox* and not decomposable, and extracts parallelism through a suitable micro-batching technique. By working without assumptions on the function, the approach is not optimal for incremental queries (for which GPU solutions already exist in the literature), but extends the space of queries that can benefit from GPU processing, which is our main goal.

C. MICRO-BATCHING CONFIGURATION

The most relevant GPU-based streaming prototypes (reviewed in Section VI) share a common approach inspired by the *micro-batching* technique of Spark Streaming [18]. They buffer input tuples in small batches, and offload the processing of each batch on the GPU. The size of each microbatch has a profound effect on the achieved throughput and response time. Its optimal value depends on information about the stream rate and the query workload, and the GPU features in term of cores, memory and interconnection bandwidth. So, any static choice of it is not portable and leaving its configuration to the users is not practical.

To overcome these issues, `Gasser` adopts an *online learning approach*. During an initial short portion of the execution after the query submission (called *calibration*), `Gasser` tries different configurations (e.g., batch sizes) and collects measurements needed to build a prediction model used to guess the performance of untried configurations. At the end of this phase, the system chooses a configuration that according to the model provides the best throughput. Notably, the calibration in our approach is fully integrated with the normal execution from the user viewpoint, that is the stream is never blocked and `Gasser` is always able to deliver results during the calibration without interruptions.

III. GASSER ARCHITECTURE

`Gasser` is written on top of `FastFlow` [19], a C++ streaming framework for multicores, and `Cuda`. To instantiate a `Gasser` operator, the programmer uses our API to construct the C++ operator object by passing as input arguments to the constructor the window specification (w, s) and the non-incremental function (through a lambda expression).

`Gasser` keeps the layout of the data structures simple and efficient (i.e. flat arrays without pointer traversal). For the sake of compatibility with older GPU models, we do not make use of the *Unified Memory* support recently introduced by NVIDIA. That is, the `Gasser` run-time system explicitly manages data transfers to/from the device. `Gasser` currently supports count-based windows and we aim to extend the support to other windowing models in the future.

Figure 1 depicts the architecture with its essential elements. The symbols `D`, `C` and $\mathcal{R}_0, \ldots, \mathcal{R}_{n-1}$ are the *distributor*, the *collector* and the $n > 0$ *replica* entities. The distributor routes input tuples (redirected by the placeholder operator in the topology, OP_x in the figure), while the replicas are functionally equivalent entities in charge of executing the processing on different sliding windows. The collector receives window results from the replicas and delivers them to the topology in increasing order of the window identifier. Such entities are executed by the runtime with threads running on the CPU cores (the number of replicas is called *concurrency level* in this paper). The replica threads, in addition to doing the pre- and post-processing tasks on the CPU, can be configured in two modalities: *1)* they can call directly the non-incremental function on each complete window and run it on a CPU core; *2)* they can batch tuples in order to collect enough

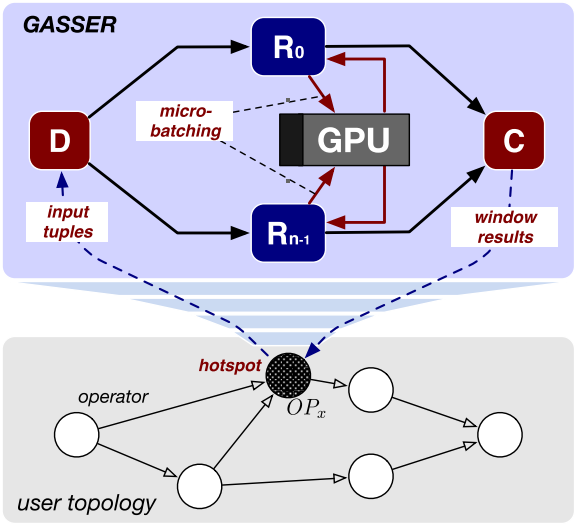

FIGURE 1. `Gasser`: **parallel window-based operator with GPU offloading (red arrows).**

data for processing several contiguous windows, whose parallel computation is offloaded on a GPU. In both cases, window results are delivered to the collector thread.

To support GPU offloading, the design follows two principles: *i)* input tuples are buffered and managed in *sliding microbatches*; *ii)* `Gasser` uses a specialized form of *data parallelism* for windowed processing to leverage the high parallelism provided by modern GPU devices.

A. SLIDING MICROBATCHES

`Gasser` is based on a general assumption that *different windows can be processed independently from each other*, condition that is verified in almost all the common sliding-window queries [5] presented in the literature so far.

In all the past approaches, the stream is partitioned into disjoint batches usually of a fixed size. In some systems the batch size depends on the specification of the sliding window (e.g., in Spark Streaming both the window length and the sliding factor must be multiple of the batch size), while in others it is kept independent of the windowing semantics like in `Saber` [6]. In `Gasser` we adopt a notion of partially overlapped (*sliding*) batches. In the GPU offloading mode, each replica continuously buffers the tuples needed to execute several consecutive windows of the stream. We denote by $\mathcal{B}_w > 0$ a configuration parameter of `Gasser` corresponding to *the number of windows per batch* (called *batch length*). Figure 2 shows microbatches (hereinafter *batches* for brevity) of $\mathcal{B}_w = 3$ windows each with $w = 4$ tuples that slides every $s = 1$ tuple. The number of tuples in a batch (called *batch size*) is $w_b = (\mathcal{B}_w - 1) \cdot s + w$ and the *batched sliding factor* is $s_b = \mathcal{B}_w \cdot s$ tuples. Batches are like macro windows of w_b tuples that slide every s_b tuples. Differently from past approaches [3], [6], in our batching scheme each window is fully contained within a batch, a condition allowing any generic non-incremental algorithm to be used with `Gasser`.

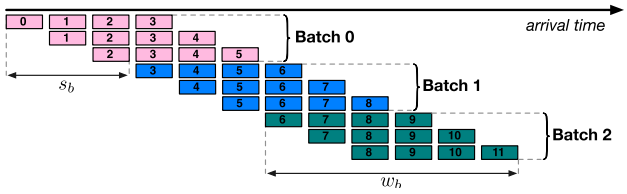

FIGURE 2. **Micro-batching with $\mathcal{B}_w = 3$ and count-based windows of length $w = 4$ and slide $s = 1$ tuples.**

Batches are logically assigned to the replicas in a round-robin fashion. In the figure, the first replica receives all the tuples of the first batch (tuples $0-5$) while the second batch is assigned to the second replica (tuples $3-8$). The three tuples shared in both the batches (tuples $3-5$) are multicasted to both the replicas. Algorithm 1 shows the pseudocode executed by the distributor at each new tuple arrival. The algorithm extracts the identifier of the tuple t and calculates the range of identifiers [*first* . . *last*] of the batches containing that tuple. Then, t is transmitted to the replicas whose identifier is in the set $\mathcal{D}_{dst}$, i.e. those replicas assigned to at least one batch in the range [*first* . . *last*]. The case with $\mathcal{B}_w = 1$ is a special one, where batching is not applied and single windows are assigned to the replicas (it is used when replicas directly compute windows on the CPU).

Algorithm 1 Batched routing strategy of the Distributor

```
1:  procedure routingTuple(t)                    ▷ t is a new input tuple
2:      if id(t) < w_b then                      ▷ id(t) extracts the identifier of t
3:          first ← 0
4:      else
5:          first ← ⌈(⌋id(t) + 1 − w_b)/s_b
6:      last ← ⌈(⌋id(t) + 1)/s_b − 1
7:      D_dst ← ∅
8:      for j ← first to last do
9:          D_dst ← D_dst ∪ {j mod n}
10:     for each r ∈ D_dst do
11:         send t to replica r
```

This micro-batching technique was used in the past to reduce the replication of tuples among replicas [20]. In this work we use it with two goals: *i)* amortize the cost of host-to-device data transfers; *ii)* find enough parallelism to exploit GPU for non-incremental queries.

B. INTER-WINDOW PARALLELISM ON GPU

Each thread executing an operator's replica in `Gasser` continuously buffers input tuples, prepares its batches by performing some pre-processing tasks (e.g., projection, data conversion) and then offloads the processing on the GPU by launching a `Cuda` kernel per batch. The `Cuda` kernel leverages the GPU parallelism degree by running the non-incremental function in parallel on each window within the batch. Figure 3 sketches the general idea. Several variants of the kernel exist and can be configured by the user. Specifically, the kernel has different options in terms of *data layout* and *batch processing* modalities. To increase parallelism,

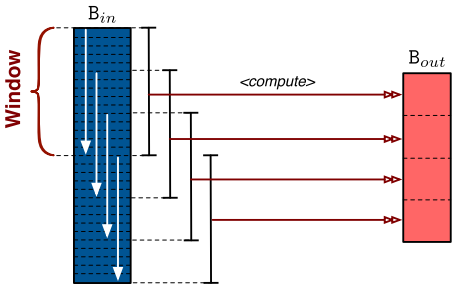

FIGURE 3. `Cuda` **kernel running over the windows in a batch. Case with** $\mathcal{B}_w = 4$ **windows per batch.**

each replica thread on the CPU has at least one dedicated `Cuda` stream to offload the processing of the batches on the GPU. A `Cuda` stream object allows a sequence of operations (e.g., copy requests, kernel executions) to be offloaded on the device in a specific order. Having more than one `Cuda` stream, at least one per replica, allows kernels and copy transfers to be performed by the device in parallel.

DATA LAYOUT. A batch is represented on the GPU as a flat array B_{in} of w_b tuples. `Gasser` supports either a *row-oriented layout* with tuples stored contiguously, or a *column-oriented layout* where B_{in} is actually composed of separated arrays each storing the same attribute of the tuples in contiguous memory. Once the kernel is launched, B_{in} is used in *read-only* mode and the results are written in an array B_{out} of $\mathcal{B}_w$ results on the device. In the default setting, `Gasser` uses a *plain copy* approach. Each replica has $k > 0$ `Cuda` streams and pairs (B_{in}, B_{out}) used in a circular manner. When a batch b is ready, the replica starts asynchronously the data transfer and the kernel on one of the `Cuda` streams using a pair (B_{in}, B_{out}). The batch is copied from scratch from the CPU to the GPU and, depending on the GPU features, the data transfer and the kernel on the next batch b' launched by the same replica will use a different stream and pair of arrays and can be overlapped with the kernel working on b.

Some of the tuples in the batch b' could have been already copied on the GPU if they took part of the previous batch b. They are exactly $w_b - s_b \cdot n$ tuples. To leverage this data sharing, `Gasser` can be configured to use an *incremental copy*, where each replica works with one (B_{in}, B_{out}) pair and `Cuda` stream, and the new tuples of b' are copied in the same B_{in} by overwriting the oldest $s_b \cdot n$ ones. However, this requires a synchronous execution of the kernels launched by the same replica, and a more complex processing since some windows may have a *wrap around* representation in the batch, with some of their tuples stored at the end and others at the beginning of B_{in}. Since this may nullify the advantage of copying less data, the plain copy semantics is the default option adopted by `Gasser`.

MICRO-BATCH PROCESSING. The kernel is organized as a 1D grid of $\mathcal{B}_w$ `Cuda` threads, thus it is essential to chose the $\mathcal{B}_w$ value properly in order to exploit the full potential of the GPU. Based on the current GPUs trend, the batch length should be in the order of hundreds or thousands of windows. The configuration in terms of threads per block will be discussed in Section V.

As depicted in Figure 3, all the `Cuda` threads execute the very same code (the non-incremental query function) in parallel on the partially overlapped data segments corresponding to the windows to compute. The data layout guarantees that: *i)* the tuples (or their attributes) are stored contiguously, *ii)* the starting tuples of two consecutive windows are very close to each other since the stride is proportional to the sliding parameter s, which is of few items in scenarios where windows trigger very frequently (the ones for which GPU processing is expected to be beneficial). If the most convenient layout is chosen for the batch (e.g., column-oriented with aligned data), and if the function has a sequential access pattern on the window data, requests issued to the global memory by sibling threads can be grouped into single requests to save memory bandwidth.

C. GASSER API

In this part, we introduce how to use `Gasser` in the context of the `FastFlow` parallel programming framework [19] written in `C++`. A `FastFlow` application is essentially a pipeline of streaming operators exchanging data items. Operators are implemented by extending the `ff_node_t` template class. Each operator executes a processing function every new data item, which is implemented by overwriting the `svc()` virtual method of the base class. To efficiently exchange data items, the run-time system adopts lock-free queues [21] used to exchange pointers to heap-allocated data structures. The data exchange is fully transparent to the high-level programmer, e.g., the pointer to the returned value of the `svc()` method is automatically pushed into the input queue used by the next operator in the pipeline.

Listing 2 shows an example of `Gasser` usage. At line 15, a `FastFlow` application consists of a pipeline (object of type `ff_Pipe`) of four operators: a generator in charge of producing data items (e.g., by reading a file or a socket), a stateless operator in charge of applying a one-to-one transformation on each input item, a `Gasser` operator of name `winOp` and a sink. The instance `winOp` is created at line 13 by providing as input arguments to its constructor the function to be executed per window (created as a host/device lambda at line 9) and by providing the basic parameters of sliding windows (i.e. length and slide). Not shown in the code snippet for the sake of space, the constructor may receive additional arguments such as a pre-processing function, to be executed per window in order to prepare the data (e.g., by projecting only the needed attributes and/or using a column-oriented representation of input data), and a post-processing function. By using such triple of pre-/compute/post- processing functions, the user can isolate the execution part that is more suitable to be offloaded on the GPU with full control on the data layout to be used.

Finally, in the example the application is run synchronously with respect to the main process by spawning a

Listing

```
1  class TransformNode: public ff_node_t<RAW_IN, IN> { ...
2     IN *svc(RAW_IN *input) { ...; return result; }
3  };
4
5  int main() {
6     ...
7     Generator gen(...);
8     TransformationNode transform(...);
9     auto winF = __host__ __device__ [] (IN *win_data, OUT *result, size_t size, ...) {
10       // code to be executed on a sliding window
11       ...
12    }
13    GASSERNode<IN, OUT, decltype(winF)> winOP(..., winF, win_len, win_slide, ...);
14    Sink sink(...);
15    ff_Pipe<RAW_IN, OUT> application(gen, transform, winOp, sink);
16    application.run_and_wait_end(); // application launched synchronously
17    return 0;
   }
```

Listing 2. Example of instantiation of a `Gasser` operator in a `FastFlow` application.

pool of threads (line 16). Automatically, the `Gasser` operator starts its calibration phase (described in the next Section) to choose the right configuration to use.

IV. AUTO-TUNING APPROACH

`Gasser` executes the non-incremental function in parallel on different windows either on the CPU cores, or can leverage the offloading on the GPU to exploit further parallelism. In the first case, it can run with $n^{max} > 0$ different configurations, equal to the maximum number of replica threads (this depends on the number of available CPU cores). In the second case, the space of configurations also includes the batch length with its possible values $\mathcal{B}_w > 0$.

Since the space of the feasible configurations $\mathcal{C}$ can be potentially large by preventing an efficient manual tuning by the user, `Gasser` is provided with an *auto-tuning support*[1] able to automatically find the best configuration. Since `Gasser` works under unknown conditions related to the arrival rate from the input stream and the computational features of the non-incremental function, this support has a double valence:

- it is used to understand whether the offloading on the GPU is really effective and to avoid using the GPU if it does not help in achieving the highest throughput (e.g., if the non-incremental function has too low warp efficiency if executed in parallel on the `Cuda` cores);
- the auto-tuning support is in charge of finding the best pair $< n, \mathcal{B}_w >$ optimizing throughput, where $\mathcal{B}_w > 1$ has sense only for configurations that offload the query processing on the GPU.

As hinted in Section II-C, we assume a stationary workload (whose mean value does not change in time) and we apply a *calibration phase* during the initial part of the execution. During this phase, the auto-tuning support tries a small set of possible configurations for a short time, by recording the achieved throughput for each of them, and predicts the performance of the untried configurations once the model becomes sufficiently accurate. Then, it configures the system with the optimal configuration for the rest of the execution.

In the rest of this section, we will focus on the two fundamental points to achieve this behavior: first we need mechanisms to change the configuration during the system execution, second we need a strategy able to decide which configurations to try and how the outcome of the untried ones can be predicted with good accuracy.

A. CALIBRATION MECHANISMS

During the calibration, the auto-tuning support seamlessly changes the current configuration several times without blocking the processing. We will focus separately on mechanisms to change the number of replicas and the batch length. In the ensuing discussion, we refer to the general case where a configuration consists of a concurrency level and a chosen batch length. For CPU-only configurations, the batch length is always equal to 1, which represents a special case.

1) CHANGING THE CONCURRENCY LEVEL

Suppose that the auto-tuning support changes the number of replicas from n to $\overline{n}$. This requires different steps: *i)* the distributor must assign batches to $\overline{n}$ replicas instead of n; *ii)* the replicas must be informed in order to do correctly the removal of the old tuples. Given a reconfiguration, we identify a special tuple defined as follows:

Definition 1 (Reconfiguration tuple): Let a reconfiguration r occur at time instant τ_r. The reconfiguration tuple t_r is the last tuple arrived before time instant τ_r.

In general, it is not possible to immediately change the distribution from the next tuple seen after t_r because there exist a number of batches that include the tuple t_r but are not complete yet, and the distribution must guarantee that the replicas that were already assigned to those batches will receive all the tuples to complete them. We present two different techniques: the first (*soft reconfiguration*) delays the application of the configuration change in order to avoid replaying any input tuple of the stream; the second (*hard reconfiguration*) anticipates the change at the expense of replaying some tuples that are needed for the correct processing by the replicas. In this part we describe the behavior of the two approaches while an experimental comparison of their efficiency will be proposed in Section V.

SOFT RECONFIGURATION. The soft reconfiguration consists of three phases starting from the reconfiguration time instant to when the new distribution is fully in operation. In the first phase, the old distribution is still used to assign the batches to the replicas. In the second phase, both the distributions are used in a mixed mode, i.e. the old one is used to assign to the replicas the batches older than a specific *switch batch*, while newer batches are assigned using the new

[1]The basic idea of our auto-tuning support is based on our prior experience with `Nornir`: http://danieledesensi.github.io/nornir/

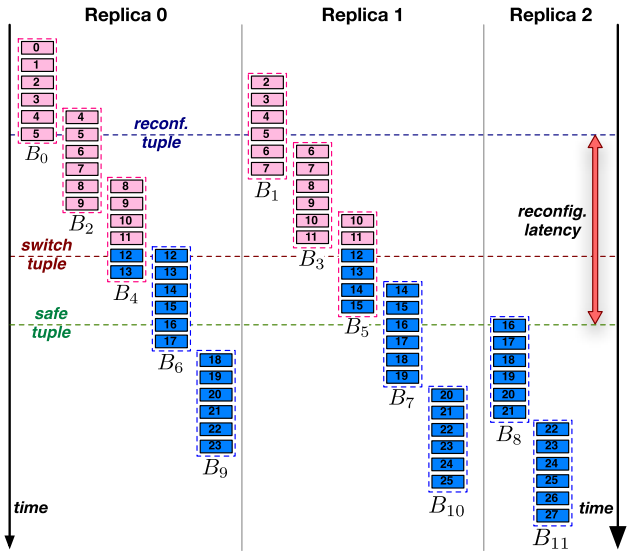

FIGURE 4. Toy example of a soft reconfiguration: the number of replicas is increased by one after tuple 5.

distribution. In the last phase, only the new distribution is used and the reconfiguration is complete.

We show a toy example in Figure 4. We assume $w = 5$ and $s = 1$ with $\mathcal{B}_w = 2$, which reflects in batches of $w_b = 6$ tuples that slide every $s_b = 2$ tuples. We suppose that before the reconfiguration, n is equal to 2 and then the auto-tuning support decides to add a replica after the arrival of tuple 5.

Batches are assigned to the two replicas in a circular manner. At the time instant of the requested change, three batches are open—B_0, B_1, B_2. However, the reconfiguration cannot be applied from the next batch B_3 because, according to the old modulo $n = 2$ distribution, this batch should be assigned to the second replica $\mathcal{R}_1$ while with modulo $\overline{n} = 3$ it should be assigned to the first replica $\mathcal{R}_0$. In the soft reconfiguration approach, we look for a way to make the two scheduling policies of batches compatible, thus we need to still assign B_3 to $\mathcal{R}_1$ and to find a seamless way to change the distribution and start using the additional replica. To this end, we define the following notion of switch tuple:

Definition 2 (Switch tuple): The switch tuple t_{sw} is the first tuple received after t_r (included) that opens a batch assigned to the first replica $\mathcal{R}_0$ with both the old (modulo n) and the new (modulo $\overline{n}$) distribution policy.

The switch tuple can be found by applying expression (1) where $id(t)$ denotes the unique identifier of tuple t and LCM is the *least common multiple* of two integer numbers:

$$id(t_{sw}) = \left\lceil \frac{id(t_r)}{\text{LCM}(n, \overline{n}) \cdot s_b} \right\rceil \cdot \text{LCM}(n, \overline{n}) \cdot s_b \qquad (1)$$

The first batch containing t_{sw} is called *switch batch*.

In the example, the switch tuple is tuple 12 that starts the batch B_6 ($6 \mod 2 = 6 \mod 3 = 0$). During the time period between the reconfiguration tuple and the switch tuple, all the batches are assigned using the old modulo n distribution. When the system receives the switch tuple, it enters in a transitory phase where tuples must be routed to the replicas (line 9 in Algorithm 1) by remembering that the batches with identifier smaller than 6 are assigned to the replicas using modulo $n = 2$, while the batches with identifier greater or equal than 6 are assigned using modulo $\overline{n} = 3$. For example, tuple 13 belongs to batches: B_4 assigned to $\mathcal{R}_0$ with the old distribution, B_5 assigned to $\mathcal{R}_1$ with the old distribution, and B_6 assigned to $\mathcal{R}_2$ with the old/new distribution. In all cases, the system knows when this transitory phase ends:

Definition 3 (Safe tuple): The safe tuple t_{sf} is the first tuple after the switch tuple whose batches are assigned to the replicas using the modulo $\overline{n}$ operation only.

The safe tuple in the example is tuple 16 and in general it is computed as follows:

$$id(t_{sf}) = id(t_{sw}) + w_b - s_b \qquad (2)$$

When the system receives the safe tuple, it starts using the new distribution only. Furthermore, in order to make the replicas able to do the removal of the old tuples correctly, a special message is sent to all the replicas immediately after the arrival of t_{sf}, in order to inform them that every time their batch slides the oldest $s_b \cdot \overline{n}$ tuples must be deleted (instead of $s_b \cdot n$). In the example of the figure, this changes from 4 to 6 tuples.

The same reasoning can be generalized to different cases such as when $\overline{n} < n$ (a decrease in the concurrency level) or for any increase/decrease of any size. One of the merits of this solution is that tuples do not need to be replayed by the distribution entity. However, the approach delays the completion of the reconfiguration until a safe point is found, and this may imply to receive a long sequence of tuples before completing the change:

Definition 4 (Reconfiguration extent): The extent $\mathcal{L}_r > 0$ of a reconfiguration r is the number of tuples received from the reconfiguration tuple (included) to the safe tuple (excluded).

Proposition 1: An upper bound to the size of the reconfiguration extent with the soft approach is $\mathcal{L}_r \leq \text{LCM}(n, \overline{n}) \cdot s_b + w_b - s_b$.

Proof: The number of tuples from the reconfiguration one (included) to the safe tuple (excluded) is $\mathcal{L}_r = id(t_{sf}) - id(t_r)$. This can be written as:

$$\mathcal{L}_r = id(t_{sw}) + w_b - s_b - id(t_r)$$

where $id(t_{sw})$ is obtained by (1). We observe that for any pair of positive numbers x, y the inequality $\lceil x/y \rceil \cdot y \leq x + y$ holds. Therefore, we can write:

$$\begin{aligned} \mathcal{L}_r &\leq id(t_r) + \text{LCM}(n, \overline{n}) \cdot s_b + w_b - s_b - id(t_r) \\ &= \text{LCM}(n, \overline{n}) \cdot s_b + w_b - s_b \end{aligned}$$

■

Since this upper bound is proportional to the LCM between the old and the new concurrency level, the number of tuples to process before completing the reconfiguration may be large and thus the reconfiguration delay.

HARD RECONFIGURATION. Making compatible the two round-robin assignments of batches to the replicas is not strictly necessary if the protocol is made a little bit more complex and anticipates the configuration change. In the hard approach we recognize two phases: in the first one the old distribution is still used to assign batches, while in the second phase we abruptly change to the new assignment policy. To identify correctly the new safe tuple, we redefine the switch tuple for the hard approach as follows:

Definition 5 (Switch tuple): The switch tuple t_{sw} is the first tuple received after t_r (included) that according to the previous assignment policy (modulo n) is the first of the next batch assigned to $\mathcal{R}_0$.

This tuple can be found using a different expression with respect to (1):

$$id(t_{sw}) = \left\lceil \frac{id(t_r)}{n \cdot s_b} \right\rceil \cdot n \cdot s_b \tag{3}$$

The safe tuple t_{sf} is defined as in (2). During the time interval between t_r and t_{sf} the distributor uses *the old assignment policy only* (modulo n). When the system receives t_{sf}, and *before* distributing it to the replicas, some actions are executed:

- the distributor entity sends a special message to all the replicas that throw away all their buffered tuples. The replicas are made aware that every time one of their batches slides, the $s_b \cdot \overline{n}$ oldest tuples must be removed;
- the distributor/replicas renumber the batches, i.e. the batch started by the safe tuple t_{sf} will be the new batch B_0;
- the distributor entity replays all the tuples from t_{sw} (included) to t_{sf}, and such tuples are distributed based on the new batch numbering using modulo $\overline{n}$.

Figure 5 shows this idea applied to the toy example. The switch tuple is tuple 8 and t_{sf} is tuple 12. At the arrival of the safe tuple, the first four batches ($B_0 - B_3$) have already been completed by $\mathcal{R}_0$ and $\mathcal{R}_1$. The special message sent by the distributor entity forces the two replicas to throw away tuples $8-11$ (by $\mathcal{R}_0$) and tuples $10-11$ (by $\mathcal{R}_1$). Then, the batches are renumbered: i.e. batch B_4 becomes B_0, B_5 becomes B_1 and so forth. In this example, we replay the tuples to the same replicas to which they were sent before (e.g., tuples 8-11 to $\mathcal{R}_0$). However, this is a particular case and tuples might be replayed to different replicas with respect to the previous distribution.

This approach allows the system to reduce the size of the reconfiguration extent which, in the hard approach, *also includes the replayed tuples received between t_{sw} and t_{sf}*.

Proposition 2: An upper bound to the size of the reconfiguration extent with the hard approach is $\mathcal{L}_r \leq s_b(n-2)+2w_b$.

The proof follows the same reasoning of the one for Proposition 1. As intuitive, the upper bound is no longer proportional to the LCM between n and $\overline{n}$ but depends on the old concurrency level n and the batch length $\mathcal{B}_w$ only.

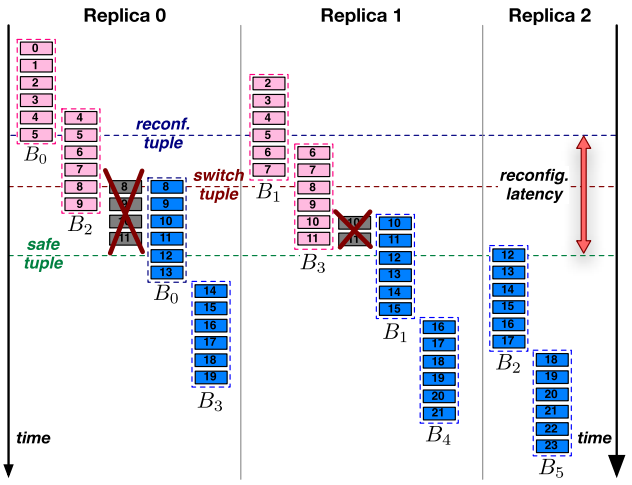

FIGURE 5. Toy example of a hard reconfiguration: number of replicas increased by one after the arrival of tuple 5.

2) CHANGING THE MICRO-BATCH LENGTH

The auto-tuning support can change the number of windows per batch from a value $\mathcal{B}_w$ to a new one $\overline{\mathcal{B}}_w$. This reflects in a different $\overline{w}_b$ and $\overline{s}_b$ to be used. We have designed a soft and a hard reconfiguration approach following the same reasoning we presented before. In the implementation of the soft approach, the switch tuple is the first one after the reconfiguration tuple that, with both the old and the new batch length, open a batch assigned to the first replica $\mathcal{R}_0$. We can note that this tuple may be very far ahead since its identifier is proportional to LCM($\mathcal{B}_w$, $\overline{\mathcal{B}}_w$). Instead, in the hard approach, the switch tuple is still computed using (3), where s_b is the batched sliding parameter with the old batch length $\mathcal{B}_w$. The special message, sent to all the replicas *before* the safe tuple, will include the new batch size $\overline{w}_b$ and the value $\overline{s}_b \cdot \overline{n}$ used by the replicas to correctly remove the old tuples. The number of replayed tuples is still $w_b - s_b$. Referring to Figure 5, a change from $\mathcal{B}_w = 2$ to $\overline{\mathcal{B}}_w = 3$ (without changing the replicas, i.e. $n = 2$) translates into a new batch size of $\overline{w}_b = 7$ tuples with $\overline{s}_b = 3$ tuples. After the replay of tuples $8 - 11$, $\mathcal{R}_0$ is in charge of computing the renumbered batch B_0 (old B_4) of 7 tuples.

It is worth noting that the hard approach, besides likely having a smaller reconfiguration extent, it is also more flexible. Indeed, it allows changing both the parameters together (i.e. the batch length and the concurrency level) since the switch tuple and the tuples to be replayed depend only on the configuration before the switching (current n and B_w). This is important in our calibration approach, since we are able to switch from a configuration to any other one in a singe step instead of splitting the change into two different reconfigurations (for the batch length first and then for the concurrency level, or vice-versa).

B. CALIBRATION STRATEGIES

The calibration strategy applies the mechanisms described before to try a small subset of configurations and then to

predict the best configuration to choose for the rest of the execution by building a prediction model. This is performed by an active entity called *autotuner*, a thread on the CPU that chooses the configurations to be tried, receives measurements from the collector and updates the model. The autotuner applies the strategy in two phases:

- *CPU calibration*: to find the best CPU-only configuration where the concurrency level is changed during the calibration phase. The autotuner remembers the best throughput achieved by using only the CPU;
- *GPU calibration*: to find the best GPU configuration by exploring a larger space of possible configurations where, in addition to the concurrency level, also the batch length is chosen in a set of discrete possible values.

Then, the autotuner applies for the rest of the execution the best configuration found among the two phases above, thus avoiding using the GPU if it is not protifable for improving throughput with the given query and data stream features.

In this part, we will study how to build the prediction model and what are the two strategies that are currently supported by `Gasser`. Although they can be used both in the CPU calibration and GPU calibration phases, in the following we mainly refer to the GPU calibration where both the concurrency level and the batch length can be changed.

1) STATISTICAL INTERPOLATION

A configuration can be represented as a point in the space (unidimensional during the CPU calibration, or a point in a two-dimensional space during the GPU calibration). Our goal is to collect the actual performance associated with some of these points, and to predict the outcome of the other points by interpolating the collected measurements.

In doing our interpolation, we would like to choose points that are equally distributed over the space. Indeed, if we collect data from two close configurations, it is likely that they behave similarly. Therefore, the second point is not adding any useful information. For this reason, we use a *low discrepancy* generator [22] (known as *quasi-random*) to decide which configurations to test. Such generators are deterministic and cover the domain more evenly than standard pseudo-random generators.

To interpolate scattered data over an unstructured grid we used a bary-centric approach, which relies on the Delaunay triangulation.[2] Before performing the interpolation, the performance measurements of the configurations at the four edges of the grid need to be collected, i.e. with the smallest and the largest batch length and concurrency level. For this reason, these will be the first four configurations tried by any calibration strategy.

2) BASIC STRATEGY

In the *basic* strategy the autotuner can be in either of two mutually exclusive states: *1) exploration state*: the next configuration in the low discrepancy ordering is chosen and applied. Once the throughput is stable, the autotuner uses the measured throughput to refine the interpolation by updating the prediction model; *2) targeting state*: the autotuner chooses a configuration that, according to the prediction model, is able to provide the *optimal throughput* (in results/sec). In case more than one optimal configuration exists, the autotuner chooses the one with the smallest number of replicas or shorter batch length. The optimal throughput is the highest one theoretically produced by the system and defined as follows:

Definition 6 (Optimal throughput): The optimal query throughput, expressed in number of windows processed per second, is equal to $\mathcal{T}^{opt} = \lambda/s$, with $s > 0$ the sliding parameter of the sliding-window query and $\lambda > 0$ the input rate.

The pseudo-code, shown in Algorithm 2 and Table 1, is continously executed by the auto-tuner thread. In the pseudocode, we use c to denote a generic configuration ($< n >$ during the CPU calibration, $< n, \mathcal{B}_w >$ in the GPU calibration). The autotuner starts in the exploration state and runs the first $\mathcal{I} \geq 4$ configurations, where $\mathcal{I}$ is a configurable parameter, collects their performance results and updates the model. Then, it uses the model built so far to look for a configuration achieving $\mathcal{T}^{opt}$ (line 19). If this does not exist, the system tries other $t_{step} > 0$ configurations and updates the model again. If the prediction model returns a configuration that is expected to achieve the optimal throughput, the system enters the targeting phase and the configuration is applied to the system. If its real measured throughput is not the optimal one, the autotuner goes back to the exploration phase (lines 13-16) and restarts the reasoning by trying other $t_{step} > 0$ configurations to refine the model.

The strategy has three main issues that must be carefully considered and solved in `Gasser`. A first problem is that once a configuration is applied, the system may need a variable amount of time to reach a stable throughput. Our solution to this problem consists of mesuring after the reconfiguration completion the real throughput level every second and maintaining a history of the last measures (few seconds are sufficient in practice). Once the *coefficient of variation*

TABLE 1. Subroutines used by Algorithm 2.

Subroutines	Description
`apply_conf.()`	The autotuner notifies the emitter thread to start the soft/hard protocol in order to apply the configuration provided as input to the subroutine.
`rcv_throughput()`	The autotuner receives from the collector thread the stable throughput measured for the running configuration.
`update_interp.()`	A new measure $Thr(x)$ is available for the running configuration c and the Delaunay triangulation is updated by taking into account the new available point $(c, Thr(c))$.
`find_optimal()`	Based on the available measures for the tried configurations, the subroutine runs the interpolation with the available points and returns (if any) a configuration that is expected to produce a throughput close to $\mathcal{T}^{opt}$.
`get_next_conf()`	Returns a new configuration not tried so far.

[2]We used the source code provided by http://rncarpio.github.io/delaunay_linterp/

Algorithm 2 Basic strategy of the Autotuner

```
1:  c ← c^init                                   ▷ initial configuration
2:  cnt ← I
3:  apply_configuration(c)
4:  targeting ← false
5:  for i ← 1 to |C| − 1 do
6:      ▷ Wait until throughput is stable
7:      thr ← rcv_throughput()
8:      if !targeting then                       ▷ exploration state
9:          update_interpolation(c, thr)
10:     else                                     ▷ targeting state
11:         if thr ≈ T^opt then
12:             break
13:         else
14:             cnt ← cnt + t_step
15:             targeting ← false
16:             update_interpolation(c, thr)
17:     ▷ Find optimal configuration
18:     if i == cnt then
19:         c^opt ← find_optimal()
20:         if c^opt ≠ nil then                  ▷ found: apply it!
21:             targeting ← true
22:             apply_configuration(c^opt)
23:         else                                 ▷ Not found, continue
24:             cnt ← cnt + t_step
25:     if !targeting then                       ▷ get the next configuration
26:         c^next ← get_next_conf()
27:         apply_configuration(c^next)
28:     i ← i + 1
```

among the recent measures is below a threshold (by default of 5%), the throughput is considered stable and its average value is transmitted to the autotuner (line 7).

Another problem is that sometimes the measured throughput can be higher than the optimal one. This may happen when the system is run with a bottleneck configuration for a while, and then a fast configuration is chosen by the strategy. In that case, it is possible to measure a higher throughput because the system drains all the enqueued tuples received when it was a bottleneck. To avoid such spurious measures, the autotuner caps them to the estimated $\mathcal{T}^{opt}$ value obtained by measuring periodically the average input rate received by the system.

The last issue is related to the calibration time. When the system is a bottleneck with all the possible configurations, the strategy tries other t_{step} configurations each time ending up in a useless exhaustive search. To limit the duration of the calibration, the autotuner stops the exploration phase if the prediction error is lower than a threshold (by default 5%). If the errors are uniformly distributed, the prediction model is already accurate and no optimal configuration exists. This condition practically holds in all the experiments we developed to test `Gasser`.

3) RAINDROP-BASED STRATEGY

The previous strategy has a potential pitfall: once a configuration reaching $\mathcal{T}^{opt}$ is found, the system remains in that configuration although the prediction errors might hide other

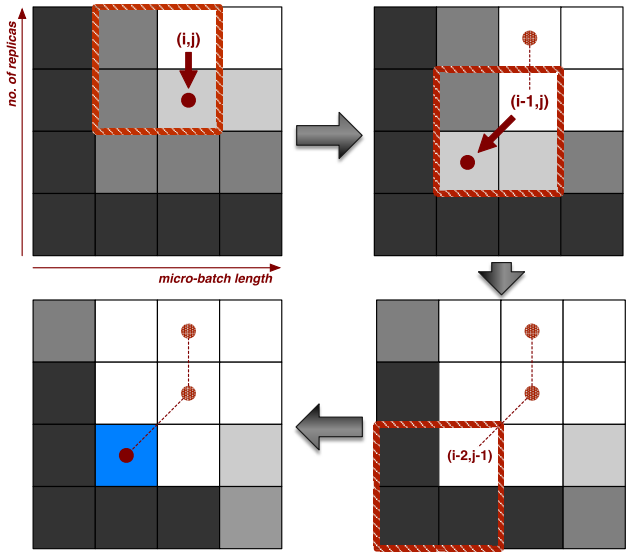

FIGURE 6. Raindrop strategy: fine movements to reach a better optimal configuration, where (i, j) is a configuration using i replicas and the j-th available batch length.

still optimal configurations using a less number of replicas (thus reducing the use of processors' cycles) or with a smaller batch length (useful to reduce latency).

To refine the strategy our idea is based on a *raindrop method*: when a raindrop falls on the ground, it flows from higher spots to lower ones due to gravity while choosing an optimum path towards a reachable lower point on the ground. To describe this analogy, we refer to the GPU calibration phase, where the space of all the feasible configurations can be represented as a matrix of points. Suppose that at the end of the targeting state the configuration (i, j) is chosen and used by the system (Figure 6). The autotuner analyzes the predicted outcomes of the configurations within a square sub-matrix of order Z whose top-right corner is (i, j). In the example we consider a 2×2 sub-matrix. The figure shows with a gray-scale the predicted throughput of each configuration, where the darker the worse.

In top-left hand side of the figure, all the configurations in the sub-matrix except (i, j) are not optimal. However, the throughput predicted for $(i-1, j)$ is slightly lower than the optimal one suggesting that the prediction error could hide an optimal configuration using fewer replicas. Therefore, the autotuner tries configuration $(i-1, j)$ and collects its real throughput. In the example this movement was effective: configuration $(i-1, j)$ actually reaches $\mathcal{T}^{opt}$ (see the top-right hand side of the figure) and the measure is used to update the prediction model. The reasoning is iterated until the movement is not effective (the measured throughput is sub-optimal) and the autotuner backtracks to the last optimal configuration found. This is what happened in the example, with $(i-2, j-1)$ the final configuration. This behavior is described in Algorithm 3, where c^{curr} is initialized with the final configuration found by the basic strategy and the subroutine `explore_neighborhood()` explores the $Z \times Z$

TABLE 2. Non-incremental queries used in the `Gasser` experimental evaluation.

Query	Description
`Q1`	The query receives a stream of financial quotes, each represented as a record of attributes such as the proposed price, volume and the stock symbol (64 bytes in total). A sliding window of the most recent quotes is continuously maintained and, when triggered, a polynomial regression is computed over all the points in the window to estimate the future price of each stock symbol and build candle-stick charts [23]. The inter-arrival times between tuples are generated using an exponential distribution with a configurable mean in order to control the input intensity of the stream.
`Q2`	The query receives a stream of sensor readings provided by the Real-time Locating System installed in the main soccer stadium of Nuremberg, Germany [24]. The dataset used for the evaluation was recorded during a training game where each player, goalkeeper, ball and referee had embedded sensors producing readings with position, velocity and acceleration values. The query maintains a sliding window of the most recent readings and computes a *K-means* clustering over the spatial coordinates of the points in the window. The goal of the query is to find the best locations in the field to place camera drones, similarly to the work done in a recent paper [25]. The original stream [24] has an average rate of $11.5K$ tuples per second. For the experimental evaluation, the stream has been accelerated multiple times to model future scenarios and increase the need of parallelism.

submatrix to look for a possible movement in the gradient descendent direction in our 2D space that still maintains an optimal throughput but with fewer replicas and/or a smaller batch (depending on the user's preference).

Algorithm 3 Raindrop-Based Refined Strategy

```
1:  end ← false
2:  c^old ← c^curr
3:  while !end do
4:      c^new ← explore_neighborhood(c^curr, Z)
5:      if c^new ≠ nil then
6:          c^old ← c^curr
7:          c^curr ← c^new
8:          apply_configuration(c^curr)
9:          thr ← rcv_throughput()
10:         update_interpolation(c^curr, thr)
11:         if thr ≯ T^opt then                    ▷ backtrack
12:             apply_configuration(c^old)
13:             end ← true
14:     else
15:         end ← true
```

At line 4 the algorithm searches in the $Z \times Z$ sub-matrix a candidate configuration to be tried. In general, Z should be small although this may require a longer path to reach the final configuration (the default value is $Z = 2$), and the candidate configuration is chosen as the one with the higher predicted throughput in the sub-matrix. Of course, due to the heuristics nature of the strategy, there could be unlucky cases where a better configuration is not reached. However, as shown in the experiments, this strategy allows `Gasser` to reach a *minimal configuration* in almost all the studied cases and with a *short calibration time*. In conclusion, we observe that one could use the raindrop strategy only (or any kind of gradient descent technique). However, having a first phase based on online learning allows us to find a maximum throughput configuration as soon as possible, and then to refine it successively with the raindrop strategy.

V. EVALUATION

The evaluation[3] is organized in two main parts. First, we show that `Gasser` is able to effectively accelerate non-incremental queries on GPUs, by outperforming the peak throughput achieved by using only CPUs for the processing. In this part, we also evaluate the cost of the generality of our blackbox model, which is expected to be less optimized for incremental queries than existing counterparts. In the second part, we show the accuracy of the auto-tuning model and we analyze in detail the two auto-tuning strategies discussed in Section IV-B and the hard/soft reconfiguration protocols.

The motivation of this work is to give GPU support for general non-incremental queries. In our evaluation, we choose two algorithms widely used in ML/Data Mining problems: a polynomial regression and a partitional clustering algorithm. They have a different arithmetic floating-point intensity (higher in the first query) while the iterative nature of the second query may generate a slight thread unbalancing among `Cuda` cores. So, they form an interesting testbed to assess the potential of our system. Table 2 gives more details by showing a possible application scenario for each query. All the experimental measures are obtained by running each experiment at least five times and collecting the average values. The standard deviation is always negligible, thus we omit to show error bars in the plots.

EXPERIMENTAL SETUP. The machine is a dual-CPU Intel Sandy-Bridge Xeon equipped with 16 cores operating at 2GHz. Each core has an L1 (32KB) and an L2 (256KB) cache. Each CPU has a L3 cache of 20MB. The machine has 32GB of RAM. The GPU, connected via PCI-e bus, is an NVIDIA K40m with 12GB of RAM and 2880 `Cuda` cores organized in 15 Streaming Multi-processors of `Cuda` 192 `Cuda` cores each. We use `gcc` compiler version `4.8.1`, `Cuda 8.0` and we turn on all the compiler optimizations (`-O3` flag). Given the features of our GPU model, each kernel is composed of $\mathcal{B}_w/384$ blocks, each one having 384 threads (two threads are executed per `Cuda` core of a Streaming Multi-processor to mask memory access latencies). For this reason, in the following we always use a batch length multiple of 384 windows: we will consider 15 different batch lengths ($i \cdot 384$ with $i = 1..15$) and 10 different concurrency levels for a total of 150 configurations (some cores on the CPU are used to execute other `Gasser` functionalities, like the stream generator and the distributor/collector threads). Furthermore,

[3]The `Gasser` source code is freely available in GitHub: https://github.com/ParaGroup/GASSER/.

Gasser is configured to use the *plain copy* approach (see Section III-B) which always gave better results in our experiments.

A. PERFORMANCE ANALYSIS

We assess when and how much the use of a GPU is convenient with respect to CPU-only counterparts. The GPU starts to be profitable if the ratio between the computation time of the window function over the time interval between two consecutive window activations is high. In those cases, batching input tuples for a short time allows gathering a large parallelism suitable for the GPU and useful to increase throughput if windows become ready to be processed faster than the rate at which they are processed by the CPU cores.

For the comparison, we use two different CPU baselines. The first is Gasser configured for using only the CPU. The second baseline is written using Apache Flink (version 1.7.1), a SPS providing good performance and generally considered better than Apache Storm [26]. The Flink implementation is based on the use of a flatmap operator in charge of adding to each input tuple the identifier of the corresponding windows in order to perform the distribution to different replicas (sub-tasks in the Flink jargon) of the window-based operator. In this way, different complete windows are executed in parallel on the CPU cores by the Flink run-time system. The query definition is provided using the non-incremental API, see Listing 1.

SUSTAINABLE INPUT RATE. Our first goal is to find the highest sustainable input rate. This limit is observed by measuring the throughput provided by the system fed by a stream with a very high rate. Then, the throughput is multiplied by the sliding factor to derive the sustainable input rate. Figures 7a and 7b show the results for the first and the second query. We tested various pairs (w, s) to study their impact. The best configurations found are reported in Table 3. As it is evident, Gasser and its offloading on a GPU device allows sustaining significantly higher input rates in all the tried scenarios. In the best case, the improvement with respect to using the CPU only is more than 10× while in the worst case it is of 2×. Complete results are in Table 3. The gain with respect to Flink (configured to use all the cores of the CPUs) is larger (from 7× to 66×), and this depends also on a completely different run-time system (Flink is written in Java/Scala and has additional features like reliability to faults not covered by our system yet).

In general, smaller values of the slide parameter mean a more frequent window activation. This is the reason for the higher gain obtained by using smaller slides, because batches become ready to be processed more frequently and the GPU is better utilized. Table 3 reports the best configurations $(n, \mathcal{B}_w)$ found.

While a large batch is important to maximize throughput, it negatively impacts the *processing latency*, i.e. the time between the instant when the triggering tuple of the window is produced by the stream's source to when the corresponding output result is delivered out from Gasser. Figure 8 shows an experiment with Q1. The slide is set to 25 tuples while we study different window and batch lengths. In this test, the input rate is chosen in order to be sure that the system is not a bottleneck in all the configurations with and without the

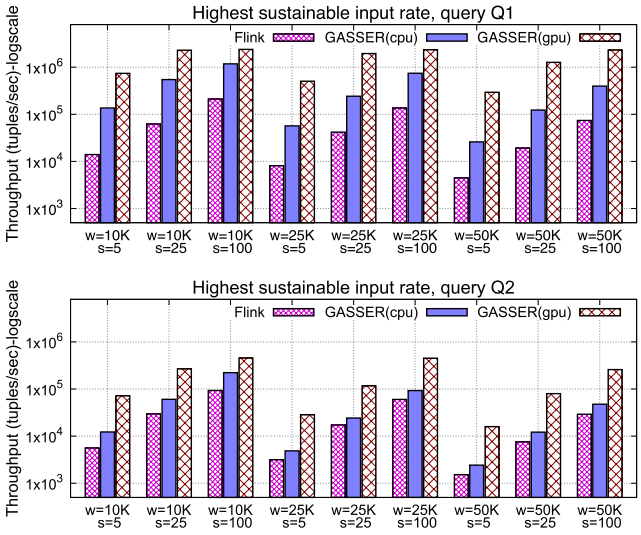

FIGURE 7. Maximum input rate (tuples/sec): comparison between Gasser (CPU-only and GPU) and Apache Flink with Q1 (a) and Q2 (b).

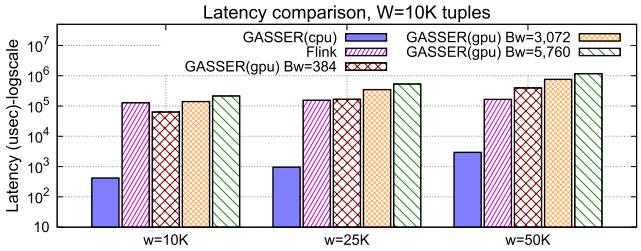

FIGURE 8. Processing latency per window: comparison between Gasser (CPU-only and GPU) and Apache Flink with Q1.

TABLE 3. Best configurations for each test case. G1 and G2 denote the gain with respect to Gasser (CPU only) and Flink.

Window setting	Query 1				Query 2			
	n	$\mathcal{B}_w$	G1	G2	n	$\mathcal{B}_w$	G1	G2
$w = 10K$ $s = 5$	7	5,376	5.4×	53×	5	1,152	5.9×	12.8×
$w = 10K$ $s = 25$	5	4,992	4.2×	37×	7	768	4.4×	9×
$w = 10K$ $s = 100$	4	768	2.0×	11×	8	768	4.3×	10.3×
$w = 25K$ $s = 5$	8	5,376	8.9×	62×	8	768	5.8×	8.9×
$w = 25K$ $s = 25$	5	4,608	8.1×	46×	5	1,152	4.8×	6.8×
$w = 25K$ $s = 100$	5	1,536	3.2×	17×	7	3,456	4.9×	7.5×
$w = 50K$ $s = 5$	10	3,840	11.3×	66×	6	1,536	6.6×	10.45×
$w = 50K$ $s = 25$	9	3,840	10.3×	56×	9	384	6.6×	10.5×
$w = 50K$ $s = 100$	4	3,456	5.8×	31×	4	3,456	5.4×	8.9×

use of the GPU, in order to prevent the latency from growing without limits during execution.

The results show that when the CPU version sustains the input rate, it is able to provide lower latency since it does not apply any batching. Furthermore, the latency on the GPU increases quickly with longer batches, which should thus be used only when this is strictly required to sustain the input pressure and the frequency of window activations. However, we point out that the latency provided by a popular SPS like `Flink` is at least comparable with the one achieved by `Gasser` with a properly sized batch length, and at least two orders of magnitude greater than `Gasser` on the CPU only.

BREAK-EVEN POINT ANALYSIS. We want to evaluate how much frequent the windows should trigger to justify the need of using a GPU. To do that, we consider a test with a stream of $1M$ tuples/sec and three different window lengths of $w = \{25K, 50K, 100K\}$ tuples. We vary the sliding factor: the smaller the slide the more frequent the windows. The *break-even point* is defined as the value of the sliding factor from which the CPU baseline has the performance of `Gasser`.

Figure 9a shows the results for query `Q1`. We measure the *GPU speedup*, i.e. the ratio of the best throughput obtained by `Gasser` to the one of the CPU baseline. For query `Q1`, the processing time per window is of some milliseconds with $w = 100K$ tuples. The break-even point (i.e. GPU speedup equal to 1) occurs for slides around $1/100$th of the window length (e.g., $s = 500$ with $w = 50K$). Figure 9b shows the results with `Q2`, which is more compute-intensive with 22.4 milliseconds to compute a window of $w = 100K$ tuples. The break-even point occurs for larger slides, i.e. thousands of tuples ($s = 2,500$ with $w = 25K$). In general, it occurs at $1/10th$ of the window length which, in this experiment, corresponds to windows triggering every few milliseconds. In those scenarios, the use of GPU contributes to increase throughput and the best GPU configuration is chosen by the auto-tuning support. Otherwise, in the less computationally-demanding scenarios (e.g., with longer slides), the GPU device is not used after the calibration.

PROFILING RESULTS. To understand how `Gasser` uses the GPU in streaming scenarios, we have collected some measures using `nvprof`, a command-line NVIDIA profiler. Figure 10 shows the results with a use case of the first query `Q1` with sliding windows of $w = 25K$ and $s = 25$. Qualitatively similar results can be achieved for the second query, not shown for brevity. The GPU occupancy (top-left plot) increases by using larger batches and so when more blocks are assigned to the Stream Multi-Processors of the GPU. However, higher occupancy does not automatically mean higher performance. Since we work on a streaming basis, greater batches need longer time to be entirely buffered, and such buffering can be useless if we have already reached the optimal throughput. In the bottom-left figure, the throughput (normalized) no longer increases with batches greater than 12 blocks, while `Gasser` is a bottleneck for smaller batches. Furthermore, as already discussed before, longer batches have negative effect on latency (top-right figure).

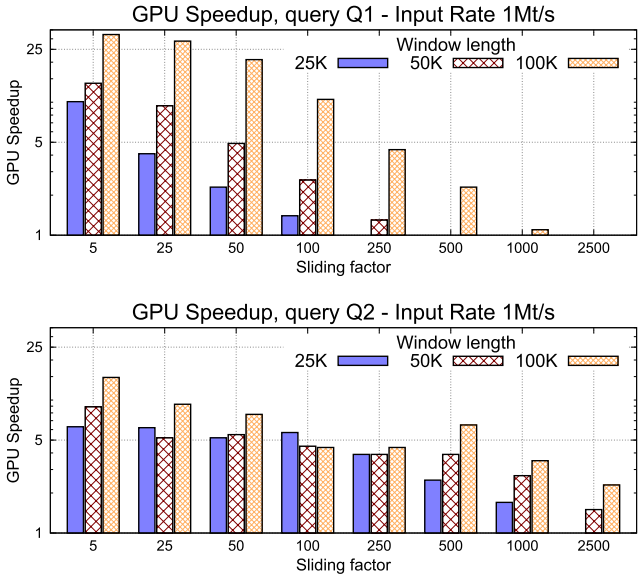

FIGURE 9. Break-even point analysis by varying the sliding factor with different window lengths: queries `Q1` (a) and `Q2` (b). GPU speedup is reported in logarithmic scale.

We report in bottom-right part, the *warp execution efficiency* metric, proportional to the probability of not-divergent branches (the higher the better). As hinted before, the second query is less efficient due to its iterative nature and represents and interesting example to understand how much a GPU is useful for queries causing a non-negligible thread divergence. Finally, we report the fraction of time spent in the `Cuda` API calls that copy batches on the device with respect to the overall kernel execution. With high frequency streams, relatively few new inputs need to be buffered and copied to extract large inter-window parallelism. For both the queries, the overhead of such calls is negligible (slightly higher for the first query, which is more fine grained).

B. PENALTY OF THE NON-INCREMENTAL APPROACH

`Gasser` targets non-incremental queries (e.g., see Listing 1). This generality in being able to use any offline algorithm for streaming cases is paid by recomputing each window from scratch. In this part, we want to evaluate the overhead paid by our approach compared with a system supporting only incremental queries and thus optimized for running them efficiently. Figure 11 reproduces the results described in [6] for `Saber`, where we use a simple incremental query that computes the average (`AVG`) of all the tuples within a window computed on a per-attribute basis. The query can be easily decomposed in two parts by calculating the `SUM` query and then dividing the value for the number of tuples per fragment. Following the same memory layout described in [6], a stream of 32-byte tuples is read at high speed from the memory where each tuple consists of a 64-bit timestamp (not used for the processing) and six 32-bit attributes drawn from a uniform distribution.

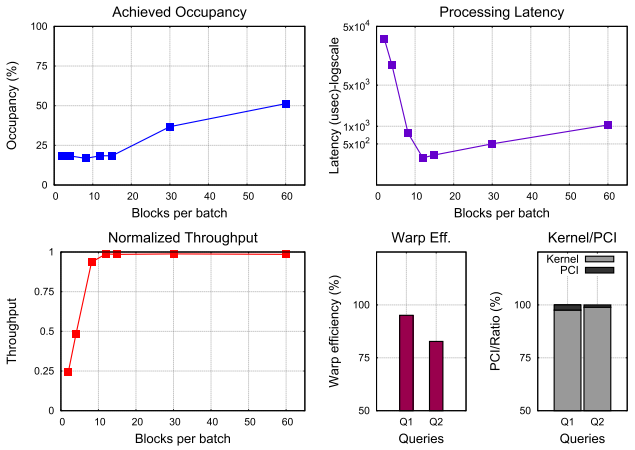

FIGURE 10. Profiling measures collected using `nvprof`.

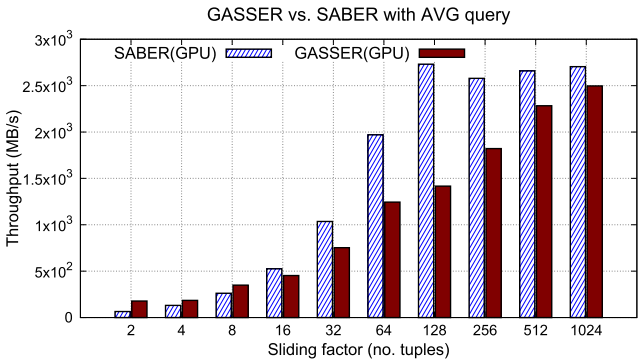

FIGURE 11. Penalty of the non-incremental processing: comparison with `Saber`.

We measure the throughput in MB/s, which represents the speed at which the stream is read from the memory and all the windows completely processed. Analogously with the experiments developed by the `Saber`'s authors, we use windows of 32KB (1K tuples) and we vary the sliding factor from two tuples (64 bytes) to the whole window length (i.e. tumbling windows). We compare the throughput provided by `Gasser` against the one of `Saber` on our machine. Since the GPU-only version of `Saber` uses only one thread on the host CPU, to be fair we run `Gasser` with one replica and using large batches (of 5, 760 consecutive windows) to fully exploit the GPU device. `Saber` is configured in its default setting, where batches are disjoint and of size 32, 768 tuples (1MB). Fig. 11 shows the result of this comparison.

With large sliding factors the gap between the non-incremental processing and the incremental approach is reduced up to 8% with tumbling windows, while it is maximum (49%) with a slide of 128 tuples. Indeed, with long slides `Saber` works with a smaller number of larger fragments and the gap is practically nullified in case of tumbling windows, where results are computed from scratch in both cases. Interestingly, with very small slides `Gasser` provides slightly better performance than `Saber`. This probably depends on the higher overhead spent in managing and assembling a greater number of fragments.

TABLE 4. Reconfiguration extent in terms of no. of tuples from the switching request until the reconfiguration completion.

		Soft		Hard	
Slide	$\mathcal{B}_w$	Avg. $\mathcal{L}_r$	Max $\mathcal{L}_r$	Avg. $\mathcal{L}_r$	Max $\mathcal{L}_r$
5	384	3.84E+04	1.35E+05	2.44E+04	3.41E+04
5	3840	3.10E+05	1.31E+06	5.98E+04	1.89E+05
5	5760	3.17E+05	1.59E+06	1.15E+05	2.73E+05
50	384	2.15E+05	9.49E+05	7.77E+04	1.91E+05
50	3840	2.89E+06	8.23E+06	5.67E+05	1.14E+06
50	5760	3.49E+06	1.67E+07	7.45E+05	1.70E+06

To complete this analysis, we mention that `Saber` actually supports a hybrid processing where both the CPU and GPU resources are exploited for the processing of fragments in a collaborative fashion. With that approach, the performance measured by `Saber` are significantly greater than using only the GPU. We do not compare directly with this support since the `Gasser` model cannot split the window processing, and CPU cores are used only for pre-processing tasks and to offload batches onto the GPU once they have been prepared. In conclusion, and as expected, `Gasser` shows to be less efficient than existing systems for decomposable queries, but it represents the only existing competitive solution for using GPUs for accelerating any general algorithm applied on sliding windows.

C. AUTO-TUNING EVALUATION

In this final part of the paper, we assess the effectiveness of the `Gasser` auto-tuning support, both focusing on the comparison between the two different implementations of our calibration mechanisms, and by studying the behavior of the two calibration strategies.

MECHANISMS EVALUATION. In this part, we compare the soft and the hard reconfiguration approaches by varying randomly the number of replicas. We executed some experiments with $w = 10K$ tuples, $s = \{5, 50\}$, input rate of $1Mt/s$ and we use three different $\mathcal{B}_w$ values. Table 4 reports the average size of the reconfiguration extent.

The reconfiguration extent is up to one order of magnitude smaller with the hard approach and thus the replayed tuples represent a small portion of the extent. This reflects in an average delay spent to complete a reconfiguration up to some seconds with the soft approach while the hard solution is always below one second. The advantage of the hard solution is greater if we randomly change the batch length (not reported for brevity). Although the system is working during the change (i.e. the delay is not a downtime), faster mechanisms should be preferred for a shorter calibration. Therefore, the hard approach is the best solution in general.

STRATEGIES AT WORK. We choose two representative scenarios of the (w, s) and (λ) parameters. In both cases, the use of the GPU device outperforms configurations that employ only the CPU. So, we focus on the GPU calibration phase which is more interesting and with a larger configuration space. Let $\mathcal{T}(n, \mathcal{B}_w)$ be for each pair $< n, \mathcal{B}_w > \in \mathcal{C}$ the

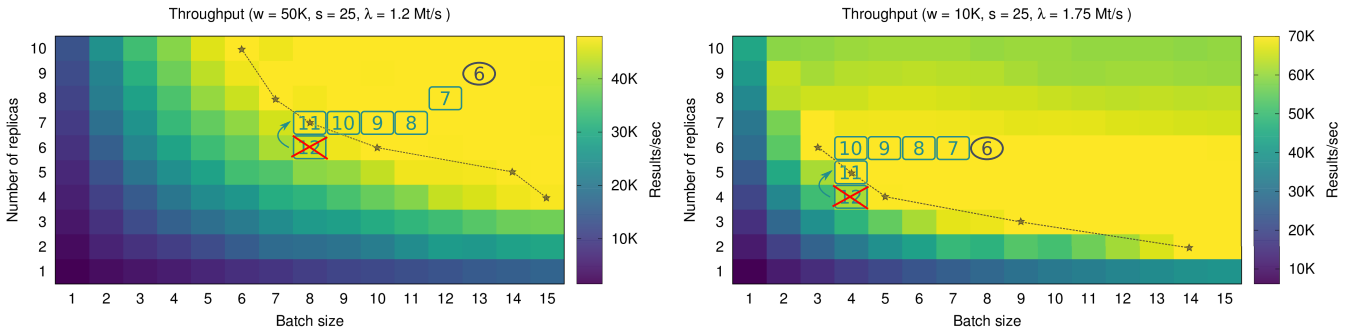

FIGURE 12. Behavior of the basic and raindrop-based strategies on two different scenarios for the first query `Q1`. The i-th batch length is equal to $i \cdot 384$ windows. $\mathcal{I}$ and t_{step} are set to the default values of 5 and 2 respectively.

real throughput measured by running that configuration. This is shown in Figure 12 as a heatmap, where darker colors are associated with a lower measured throughput.

Figure 12(left) shows the first scenario with $\mathcal{T}$ increasing by moving to the top-right corner of the domain, that is by using the greatest batch length and the higher concurrency level. The circle ⑥ indicates the configuration found by the basic strategy, and the number in the circle (six) is the number of configurations tried (five in the exploring phase and one in the targeting phase). The square boxes are the configurations visited by the raindrop-based strategy, which moves from ⑥ and ends up reaching 11. The number in the box still represents the $i-th$ configuration visited by the strategy, e.g., configuration 7 is the one visited after ⑥. When the strategy tries the configuration 12, it realizes that 12 does not give the optimal throughput and backtracks to 11 that is the last optimal configuration found.

The figure reports using a black line the *Pareto frontier*, i.e. the set of minimal configurations achieving $\mathcal{T}^{opt}$. As we can see, 11 is on the frontier. The same idea is shown in Figure 12(right) for the second scenario. In this case, the configuration in the top-right corner is sub-optimal and the maximum is in the central part of the surface. This occurs since using too many replicas and a very high rate the distributor `D` is not fast enough to efficiently distribute tuples to the replicas. Even in this case the raindrop-based strategy selects a configuration on the Pareto frontier.

PREDICTION ACCURACY. We run experiments of the two queries by computing the average relative error of the predicted throughput $\hat{\mathcal{T}}$ against the real one $\mathcal{T}$. Figure 13 shows the error under six scenarios in terms of (w, s) and input rate λ and by varying the number of tried configurations T_c to train the model. The scenarios are chosen to cover both shapes previously outlined. The second shape arises in the scenarios using $\lambda = 1.75M$ tuples/sec. The error is below 10% after training the model with 10 configurations. The strategy visited only 6% of the total configuration space.

OPTIMALITY AND CALIBRATION TIME. We study two distinct aspects for the query `Q1` and `Q2`. We consider six scenarios in term of (w, s) and λ where both the two shapes of $\mathcal{T}$ are tested. We measure the *Chebyshev distance* $d \geq 0$ from

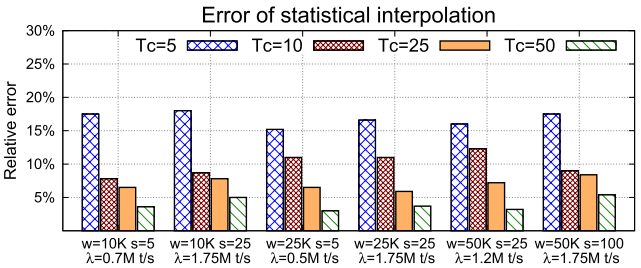

FIGURE 13. Relative error and no. of tried configurations.

TABLE 5. Optimality and calibration time (seconds).

				Basic		Raindrop	
	w	s	λ	d	C_T	d	C_T
Query Q1	$10K$	5	$0.7M$ t/s	2.00	26.3s	0.00	42.2s
	$10K$	25	$1.75M$ t/s	3.00	25.9s	0.00	45.6s
	$25K$	5	$0.5M$ t/s	0.00	34.0s	0.00	37.1s
	$25K$	100	$1.75M$ t/s	4.00	38.7s	1.33	57.5s
	$50K$	25	$1.2M$ t/s	3.00	42.4s	0.67	67.0s
	$50K$	100	$1.75M$ t/s	2.00	44.2s	0.47	59.5s
Query Q2	$10K$	5	57.5 Kt/s	2.00	80.4s	1.00	100.2s
	$10K$	25	230 Kt/s	3.00	85.3s	0.00	110.9s
	$25K$	5	23 Kt/s	3.00	179.3s	0.00	214.8s
	$25K$	100	345 Kt/s	2.00	165.1s	0.70	187.3s
	$50K$	25	46 Kt/s	3.00	351s	1.30	442.8s
	$50K$	100	172.5 Kt/s	3.00	371s	0.30	463.3s

the final configuration found by the strategy to the closest one on the Pareto frontier (i.e. it is the minimum number of moves between the two points). The smaller the distance the better is the ability of the strategy to find an optimal configuration with minimum concurrency level or batch length. We observe that the distance is sometimes not a integer because we average it on multiple runs. Table 5 shows the results.

In all the tested cases, the raindrop-based strategy ends up in an optimal configuration on the Pareto frontier or very close to it. We show also the calibration time C_T, i.e. the duration of `Gasser` calibration. In general, few configurations are tested (on average 5 with the basic strategy, $7-14$ with the refined one), and C_T is in the order of tens of seconds (at most few minutes in the worst case). This is a small portion of the execution time for long-running applications and definitely smaller than the time needed to exhaustively try all the configurations, which lasts hundreds of seconds in

the best case (e.g., it is about 800 seconds for Q1 and 3400 secs for Q2 with $w = 50K$ and $s = 100$).

VI. RELATED WORK

In this section, we review the recent papers most closely related to GPU-based supports in contexts of data stream processing systems and applications.

Saber [6] is a GPU-based framework written in Java, which proposes an execution model similar to Gasser. Saber uses a notion of batch independent of the window definition, where batches are disjoint sets of tuples and, differently from Spark Streaming, the window length and slide parameters may not be a multiple of the batch size. Batches may contain fragments of different windows, and Saber handles those fragments by computing partial results that will be used to build results of complete windows. Therefore, it supports only some operators and, more specifically, relational algebra operators (e.g., aggregation and joins) and not generic non-incremental algorithms like in Gasser. For this reason, Gasser should be used complementary to Saber since it is less optimized for incremental queries but extends the processing on GPU for queries not supported by Saber. The same batching approach is also adopted with the *bucket* operator in [27]. Instead, Gasser is based on *sliding batches*, where each batch contains the same number of windows that are computed in parallel by the GPU cores. Sliding batches have also been used in [28] to reduce the communication overhead, although for multi-core CPUs only.

G-Storm [3] extends Apache Storm with support for GPUs. G-Storm introduces the concept of *GPU-Bolt*. Like our system, to run an operator on the device the user has to write the kernel functions. G-Storm relies on JCUDA to have easy access to a GPU in Java. The execute() method of GPU-Bolts buffers incoming data in fixed-size non-overlapped batches. The merit of this work is to have extended a widely used framework. However, the system does not support sliding window since windows span across several batches. It can be used with tumbling windows which, in general, trigger less frequently than sliding windows and are less computationally demanding. However, G-Storm could be modified to extend the window support by integrating several of the Gasser concepts.

GStream [4] targets GPU clusters by hiding to the user the complexity of managing data transfers among GPU devices. Furthermore, and analogously to Gasser, it is a C++ template library allowing the creation of dataflow graphs of filters connected by stream edges. Filters process batches of input data obtained from an elastic API able to maintain the integrity of the input buffer in presence of concurrent pops by multiple downstream filters. A filter can offload the processing of each window on the GPU provided that the user is able to extract enough data parallelism from the code to be executed. This approach can be adopted when the window computation can be easily parallelized (e.g., aggregates based on associative functions), while Gasser supports GPU parallelism for any windowed operator where the provided function is treated as a blackbox. Finally, other papers are tailored for specific operators (e.g., band-join [29] and outliers detection [30]) and lack of general support to broader classes of operators.

VII. CONCLUSIONS AND FUTURE WORK

Gasser is a system for accelerating streaming operators on GPU devices. It targets non-incremental functions and provides an auto-tuning approach to automatically optimize the query throughput. It fills a gap in the literature because non-incremental queries have been overlooked in the design of previous approaches, which are instead optimized for queries allowing incremental processing. Although non-incremental algorithms are not generally the best approach in streaming contexts, existing SPSs support them to reuse well-known algorithms and legacy code. Therefore, they have a significant practical relevance.

In the future we aim at extending our work. First, to support time-based windows with variable cardinality. Second, to study Gasser and its auto-tuning approach in case of highly irregular workloads, to assess whether the proposed approach can be periodically re-executed to adapt the query configuration to time-varying input rates.

REFERENCES

[1] G. Han, M. Guizani, J. Lloret, S. Chan, L. Wan, and W. Guibene, "Emerging trends, issues, and challenges in big data and its implementation toward future smart cities," *IEEE Commun. Mag.*, vol. 55, no. 12, pp. 16–17, Dec. 2017.

[2] 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. (ICDE)*, Apr. 2017, pp. 659–670.

[3] Z. Chen, J. Xu, J. Tang, K. Kwiat, C. Kamhoua, and C. Wang, "GPU-accelerated high-throughput online stream data processing," *IEEE Trans. Big Data*, vol. 4, no. 2, pp. 191–202, Jun. 2018.

[4] Y. Zhang and F. Mueller, "GStream: A general-purpose data streaming framework on GPU clusters," in *Proc. Int. Conf. Parallel Process.*, Sep. 2011, pp. 245–254.

[5] B. Gedik, "Generic windowing support for extensible stream processing systems," *Softw., Pract. Exper.*, vol. 44, no. 9, pp. 1105–1128, Sep. 2014. doi: 10.1002/spe.2194.

[6] A. Koliousis, M. Weidlich, R. C. Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proc. SIGMOD Int. Conf. Manage. of Data*, New York, NY, USA, 2016, pp. 555–569. doi: 10.1145/2882903.2882906.

[7] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*, 1st ed. New York, NY, USA: Cambridge Univ. Press, 2014.

[8] S. Babu and J. Widom, "Continuous queries over data streams," *ACM Sigmod Rec.*, vol. 30, no. 3, pp. 109–120, Sep. 2001. doi: 10.1145/603867.603884.

[9] X. Zhao, S. Garg, C. Queiroz, and R. Buyya, "A taxonomy and survey of stream processing systems," in *Software Architecture for Big Data and the Cloud*, I. Mistrik, R. Bahsoon, N. Ali, M. Heisel, and B. Maxim, Eds. Boston, MA, USA: Morgan Kaufmann, 2017, ch. 11, pp. 183–206. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128054673000119

[10] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl, "Cutty: Aggregate sharing for user-defined windows," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, New York, NY, USA, 2016, pp. 1201–1210. doi: 10.1145/2983323.2983807.

[11] J. Krämer and B. Seeger, "Semantics and implementation of continuous sliding window queries over data streams," *ACM Trans. Database Syst.*, vol. 34, no. 1, pp. 4:1–4:49, Apr. 2009. doi: 10.1145/1508857.1508861.

[12] L. Chen and G. Lin, "Extending sliding-window semantics over data streams," in *Proc. Int. Symp. Comput. Sci. Comput. Technol.*, vol. 2, Dec. 2008, pp. 110–113.

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

[14] K. Tangwongsan, M. Hirzel, and S. Schneider, "Low-latency sliding-window aggregation in worst-case constant time," in *Proc. 11th Int. Conf. Distrib. Event-Based Syst. (DEBS)*, New York, NY, USA, 2017, pp. 66–77. doi: 10.1145/3093742.3093925.

[15] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis, "SlickDeque: High throughput and low latency incremental sliding-window aggregation," in *Proc. 21th Int. Conf. Extending Database Technol. (EDBT)*, Vienna, Austria, Mar. 2018, pp. 397–408. doi: 10.5441/002/edbt.2018.35.

[16] J.-C. Lamirel, R. Mall, and M. Ahmad, "Comparative behaviour of recent incremental and non-incremental clustering methods on text: An extended study," in *Modern Approaches in Applied Intelligence*, K. G. Mehrotra, C. K. Mohan, J. C. Oh, P. K. Varshney, and M. Ali, Eds. Berlin, Germany: Springer, 2011, pp. 19–28.

[17] C. Giraud-Carrier, "A note on the utility of incremental learning," *AI Commun.*, vol. 13, no. 4, pp. 215–223, 2000. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0034499376&partnerID=40&md5=4adedce94d99b3cdf4ed1b0df79cf0c1

[18] Z. Nabi, *Pro Spark Streaming: The Zen of Real-Time Analytics Using Apache Spark*, 1st ed. Berkely, CA, USA: Apress, 2016.

[19] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High-Level and Efficient Streaming on Multicore*. Hoboken, NJ, USA: Wiley, 2017, ch. 13, pp. 261–280. doi: 10.1002/9781119332015.

[20] C. Balkesen and N. Tatbul, "Scalable data partitioning techniques for parallel sliding window processing over data streams," in *Proc. 8th Int. Workshop Data Manage. Sensor Netw.*, Seattle, WA, USA, Aug. 2011.

[21] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Berlin, Germany: Springer, 2012. doi: 10.1007/978-3-642-32820-6_65.

[22] W. J. Morokoff and R. E. Caflisch, "Quasi-random sequences and their discrepancies," *SIAM J. Sci. Comput.*, vol. 15, no. 6, pp. 1251–1279, 1994.

[23] T. De Matteis and G. Mencagli, "Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing," in *Proc. 21st ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*, New York, NY, USA, 2016, pp. 13:1–13:12. doi: 10.1145/2851141.2851148.

[24] C. Mutschler, H. Ziekow, and Z. Jerzak, "The DEBS 2013 grand challenge," in *Proc. 7th ACM Int. Conf. Distrib. Event-Based Syst. (DEBS)*, New York, NY, USA, 2013, pp. 289–294. doi: 10.1145/2488222.2488283.

[25] X. Wang, A. Chowdhery, and M. Chiang, "Networked drone cameras for sports streaming," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2017, pp. 308–318.

[26] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, "A performance comparison of open-source stream processing platforms," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2016, pp. 1–6.

[27] M. Pinnecke, D. Broneske, and G. Saake, "Toward GPU accelerated data stream processing," in *Proc. GvD*, 2015, pp. 78–83.

[28] R. Mayer, A. Slo, M. A. Tariq, K. Rothermel, M. Gräber, and U. Ramachandran, "SPECTRE: Supporting consumption policies in window-based parallel complex event processing," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, New York, NY, USA, 2017, pp. 161–173. doi: 10.1145/3135974.3135983.

[29] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner, "The HELLS-join: A heterogeneous stream join for extremely large windows," in *Proc. 9th Int. Workshop Data Manage. New Hardw. (DaMoN)*, New York, NY, USA, 2013, pp. 2:1–2:7. doi: 10.1145/2485278.2485280.

[30] C. HewaNadungodage, Y. Xia, and J. J. Lee, "GStreamMiner: A GPU-accelerated data stream mining framework," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, New York, NY, USA, 2016, pp. 2489–2492. doi: 10.1145/2983323.2983341.

TIZIANO DE MATTEIS is currently a Postdoctoral Researcher in parallel computing. He is also a Postdoctoral Researcher with the Parallel Programming Models Group, Department of Computer Science, University of Pisa. His principal research interests include parallel and distributed computing with a particular focus in parallel data stream processing, high-level parallel programming, and energy awareness in parallel computing.

GABRIELE MENCAGLI is currently an Assistant Professor with the Computer Science Department, University of Pisa, Italy. He has coauthored over 50 peer-reviewed papers appeared in international conferences, workshops, journals, one book. His research interests include the areas of parallel and distributed systems and data stream processing. He is an Editorial Board Member of *Future Generation Computer Systems* (Elsevier) and *Cluster Computing* (Springer).

DANIELE DE SENSI is currently a Postdoctoral Researcher with the Computer Science Department, University of Pisa, Italy. His doctoral work is focused on autonomic and power-aware runtime solutions for parallel applications. He has designed algorithms to enforce power consumption and performance requirements on parallel applications through dynamic reconfigurations, by exploiting online learning techniques. He is also interested in parallel programming models, network processing applications, and HPC interconnection networks.

MASSIMO TORQUATI is currently an Assistant Professor in computer science with the University of Pisa, Italy. He has published over 90 peer-reviewed papers in conference proceedings and international journals, mostly in the fields of parallel and distributed programming and run-time systems for parallel computing platforms. He has been involved in several Italian, EU, and industry-supported research projects. He is currently the maintainer and the main developer of the Fastflow parallel programming framework.

MARCO DANELUTTO is currently a Professor with the Department of Computer Science, University of Pisa, Italy. His main research interests include the fields of parallel programming models, in particular in the areas of parallel design patterns and algorithmic skeletons. He has authored over 150 papers appearing in refereed international journals and conferences. He has been and is currently responsible for the University of Pisa research unit in different EU funded projects (CoreGRID, GRIDcomp, ParaPhrase, REPARA, and RePhrase). He is currently responsible for the EuroMicro PDP Conference series, and in the past he has been member of the Euro-Par Steering Committee.