Hardware-Conscious Stream Processing: A Survey

Shuhao Zhang[1], Feng Zhang[2], Yingjun Wu[3], Paul Johns[4], Bingsheng He[1]
[1]National University of Singapore, [2]Renmin University of China,
[3]IBM Almaden Research Center, [4]Nanyang Technological University

ABSTRACT

Data stream processing systems (DSPSs) enable users to express and run stream applications to continuously process data streams. To achieve real-time data analytics, recent researches keep focusing on optimizing the system latency and throughput. Witnessing the recent great achievements in the computer architecture community, researchers have started investigating the potential of hardware-conscious stream processing by better utilizing modern hardware capacity in DSPSs. In this paper, we conduct a systematic survey of recent works, particularly along the following three directions 1) computation optimization, 2) communication optimization, and 3) query deployment in DSPSs. Finally, we also advice potential future research directions.

1. INTRODUCTION

Despite the massive effort devoted to big data research, many challenges remain. A large volume of data is generated in real-time or near real-time and has grown explosively in the last few years. For example, IoT (Internet-of-Things) organizes billions of devices around the world that are connected to the internet through sensors or Wi-Fi. IHS Markit forecasts [8] that 125 billion such devices will be on service by 2030, up from 27 billion last year. With the proliferation of such high-speed data sources, numerous data-intensive applications are deployed in real-world use cases exhibiting latency and throughput requirements that cannot be satisfied by traditional batch processing models.

Data stream processing system (DSPS) is a software system that allows users to efficiently run stream applications, which continuously analyzes data in real-time. For example, modern DSPSs [1, 2] are able to achieve very low processing latency in the order of milliseconds. Due to its unique characteristics, and leading enterprises such as SAP [100], IBM [42], Google [12] and Microsoft [22].

Despite the successes in the last several decades, the more radical performance demand, complex analytics, as well as intensive state access in emerging stream applications [25, 97, 70] are challenging existing DSPSs. Meanwhile, great achievements have been made in the computer architecture community, which have attracted recent attentions of investigating the potential of *hardware-conscious DSPSs*, which aim to exploit the potential of accelerating stream processing on modern hardwares [22, 49, 65, 28, 51, 78, 49].

Fully utilizing hardware capacity is notoriously challenging, and a large number of studies were proposed in recent years (e.g., [99, 97, 61, 60]). This paper hence aims at presenting a systematic review of prior efforts on hardware-conscious stream processing, particularly along the following three directions: 1) computation optimization, 2) communication optimization, and 3) query deployment. We aim to show what have been achieved and reveal what have been largely overlooked. We hope that, this survey will shed light on the hardware-conscious design and architecture of future DSPSs.

Organizations. The remaining sections are organized as follows. We discuss background of DSPSs in Section 2, and present an overview of hardware-conscious stream processing in Section 3. Then, we perform an in-depth exploration of the challenges and techniques, along three main directions – computation (Section 4), communication (Section 5), and query deployment (Section 6). We conclude this survey in Section 7 with suggestion of future directions.

2. BACKGROUND

In this section, we introduce backgrounds of DSPS, including its common APIs and runtime architectures.

2.1 Common APIs

DSPS needs to provide a set of APIs for users

to express their stream applications. Most modern DSPSs such as S4 [66], Storm [2], Flink [1] express a streaming application as a directed acyclic graph (DAG), where nodes in the graph represent operators, and edges represent the message passing between operators. Figure 1(a) illustrates *word count* (WC) as an example application containing five operators. A detailed description of more stream applications can be found in [99].

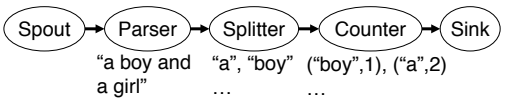

Figure 1: A stream processing example of word count (WC).

Earlier DSPSs (e.g., Storm [2]) often required users to implement each operator on their own. recent efforts from Saber [49], Flink [1], Spark-Streaming [95] and Trident [58] aim to provide higher level functional APIs (e.g., SQL) with rich built-in operators such as stream aggregation and stream joins.

2.2 Common Runtime Architectures

Modern stream processing systems can be generally categorized based on their processing models including Continuous Operator (CO) model and Bulk Synchronous Parallel (BSP) model.

Continuous Operator Model: Under the CO model, the execution runtime treats each operator (vertex of the DAG) as a single execution unit (e.g., a Java thread) and multiple operators communicate through message passing (edge of the DAG). For scalability, each operator could be executed independently in multiple threads, where each processes sub-stream of input events with stream partitioning [48]. This allows users/DSPSs to control the parallelism of each operator in a fine-grained manner [99]. Such design is adopted by many DSPSs such as S4 [66], Storm [2], Heron [3], Seep [27], and Flink [1] for its advantage of low processing latency. Other recent hardware-conscious DSPSs adopt CO model including TerseCades [70].

Bulk-Synchronous Parallel Model: Under BSP model, input stream data is explicitly grouped into micro-batches before evaluation. Each data in a micro-batch is independently processed by the entire DAG of operators (ideally by the same thread) without any cross-core communication. However, the DAG may contain synchronization barrier, where threads need to exchange their intermediate results (i.e., data shuffling). Taking WC as an example, `Splitter` needs to ensure that the same word is always passed to the same `Counter` operator. Hence, a data shuffling operation is required before `Counter`. Spark-streaming [95, 90], Google Dataflow [13] with FlumeJava [21] adopt the BSP model. Other recent hardware-conscious DSPSs adopt BSP model including Trill [22], Saber [49], and StreamBox [61].

The fundamental difference between the two processing models lies in their different evaluation flow [86]: data (i.e., CO model) or control flow (i.e., BSP model). Although there have been recent efforts at comparing different DSPSs under different models [97], it is still inconclusive which model is more suitable for utilizing modern hardware. Each model comes with its own advantage and disadvantage. For example, the BSP model naturally minimizes communication among operators inside the same DAG, but its single centralized scheduler has been identified with scalability limitation [90]. Moreover, its unavoidable data shuffling also brings communication overhead among operators. CO model provides fine-grained control (i.e., different operators can have different parallelism and placement configurations) but potentially incurs higher communication cost. However, the limitations of both model can be potentially addressed. For example, cross-operator communication overhead (under both CO and BSP models) can be overcome by exploiting high bandwidth memory [60, 74], data compression [70], infiniband [43], and architecture-aware query optimization (e.g., [97]).

3. SURVEY OUTLINE

The hardware architecture is evolving fast and provides much higher processing capability than what classical DSPSs were originally designed for. For example, recent *scale-up* servers can accommodate hundreds of CPU cores and terabytes of memory [4] providing abundant computing resources. Emerging network technologies such as Remote Direct Memory Access (RDMA) and 10Gb Ethernet significantly improve system ingress rate making I/O no longer a bottleneck in many practical scenarios [61, 25]. However, prior studies [99, 97] have shown that existing data stream processing system (DSPSs) severely underutilize hardware resources due to the unawareness of the underlying complex hardware micro-architectures.

As summarized in Table 1, we are witnessing a revolution in the design of DSPSs that exploits

Table 1: Summary of the surveyed works

Research Dimensions	Key Concerns	Related Work
Computation optimization	Synchronization overhead, cross-core communication overhead, work efficiency	CellJoin [28], FPGAJoin [51, 75], Handshake join [87, 76], PanJoin [69], HELLS-join [47, 46], Aggregation on GPU [49], Aggregation on FPGA [68, 29], Hammer Slide [88], StreamBox [61], Parallel Index Join [79]
Communication optimization	Time and space efficiency, data locality, memory footprint	Batching [99], Stream with HBM [60, 74], TerseCades [70], Stream over Infiniband [43], Stream on SSDs [54], NVM-aware Storage [72]
Query deployment	Operator interference, elastic scaling, power constraint	Orchestrating [52, 26], StreamIt [20], SIMD [41], BitStream [11], Streams on Wires [63], HRC [91], RCM [92], CMGG [67], GStream [102], SABER [49], Trill [22]

emerging hardware capability, particularly along the following three dimensions.

1) Computation Optimization: Contrast to conventional DBMSs, there are two key features in DSPSs that are both fundamental to many stream applications and computationally expensive: *windowed operator* [40] (e.g., sliding window stream join) and *deterministic operator* [16] (i.e., guarantee processing order). The support for those expensive operators is becoming one of the major requirement for modern DSPSs and is treated as one of the key dimension in differentiating modern DSPSs. Prior approaches making use of heterogeneous architectures (e.g., GPUs and Cell processors) [28, 49], multicore architectures [88, 61], and Field Programmable Gate Arrays (FPGAs) [87, 76, 51, 75, 68, 29] for accelerating those expensive operations.

2) Communication Optimization: Message passing among operators (i.e., in-core communication [18]) is often a major source of overhead in stream processing. Recent works have revealed that the overhead due to the message passing design is significant, even without TCP/IP network stack [99, 97]. Subsequently, researches have been conducted on improving the efficiency of data grouping (i.e., output stream shuffling among operators) using High Bandwidth Memory (HBM) [60], compress data in transmission with hardware accelerators and computation over compressed data [70], and leverage on Infiniband for faster data flow [43]. Having said that, there are also cases, where the application needs to temporarily store data in external storage [89] (i.e., out-of-core communication [18]). Examples include stream processing with large window operation (i.e., workload footprint larger than memory capacity) and stateful stream processing with high availability (i.e., application states have to be persistently kept). To relive the potential bottleneck caused by I/O, recent works have investigated how to achieve more efficient out-of-core communication leveraging on SSD [54] and NVRAM [72].

3) Query Deployment: At an even higher point of view, researchers have studied launching a whole stream application (i.e., a query) into varies hardware architectures. In particular, many algorithms and mechanisms have been developed to allocate/schedule operators of a stream application into physical resources in order to achieve a certain optimization goal [38]. To take advantage of modern hardware, prior works have exploited cache-conscious strategies [11], multicore [52, 33, 22], FPGA [63], and GPU [91, 92, 67, 102]. Recent works have also looked into supporting hybrid computing architectures [49].

System Design Requirements. In 2005, Stonebraker et al. [82] outline 8 requirements of real-time data stream processing. Since then, tremendous improvement has been made thanks to the great efforts from both industry and research community. In the following, we summarize existing HW-conscious optimization techniques in DSPSs by referencing to those requirements at the end of each section.

4. COMPUTATION OPTIMIZATION

In this section, we review the literature on accelerating computationally expensive streaming operators using modern hardware.

4.1 Windowed Operators

In stream applications, the processing is mostly performed using long-running queries known as continuous queries [15]. To handle potentially infinite data streams, continuous queries are typically limited to a sliding window that limits the number of tuples to process at any point in time. The window can be defined based on the number of tuples (count based) or as a function of time (time based). Windowed stream joins and sliding window aggregation are two well-known expensive windowed operators in stream processing. In the

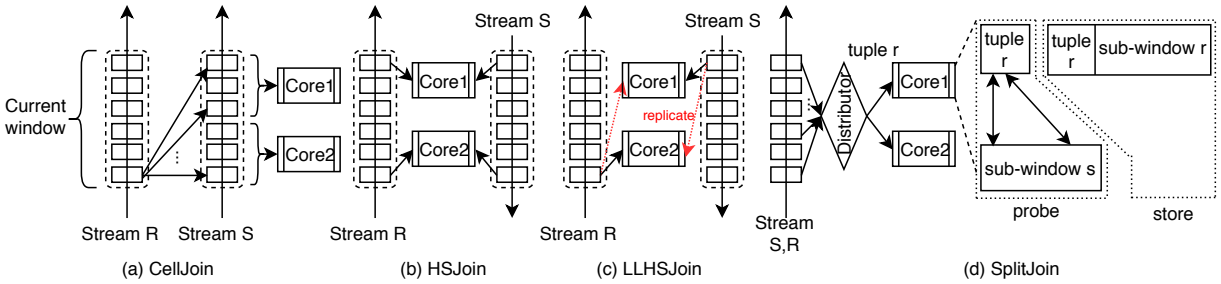

Figure 2: HW-conscious stream join algorithms (using dual-core as an example).

following, we review their algorithm overview and HW-Conscious optimizations in details.

4.1.1 Windowed Stream Joins

A common operation used in many analytical workloads is to *join* different data sources. Different from traditional join [37], which processes a large batch of data at once, stream join has to produce results on the fly [44, 81, 36, 31]. By definition, the stream join operator performs join over infinite streams. In practice, streams are cut into finite slices/windows [93]. In a two-way stream join, tuples from the left stream (R) are joined with tuples in the right stream (S) when the specified key attribute matches and the timestamp of tuples from both streams falls within the same window.

Algorithm Overview. Kang et al. [44] described the first streaming join implementation, as detailed below. For each new tuple r of stream R,

1. Scan the window associated with stream S and look for *matching* tuples.
2. *Invalidate* old tuples in both windows.
3. *Insert* r into the window of S.

The costly nature of stream join operators and the stringent response time requirements of stream application has created significant interest in accelerating stream join operators.

HW-Conscious Optimizations. Multicore processors that provide high processing capacity are ideal for executing costly windowed stream operators. However, fully exploiting the potential of a multicore processor is often challenging, due to complex processor micro-architecture, deep cache memory subsystem and the unconventional programming model in general. Figure 2 illustrates four representative works on accelerating windows-based stream joins, described in detail below.

CellJoin: An earlier work from Gedik et al. [28], called CellJoin, attempt to parallelize stream join on Cell processor – a heterogeneous multicore architecture. CellJoin generally follows Kang's [44] three-step algorithm. However, it re-partitions S, and each resulting partition is assigned to an individual core. In this way, the matching step can be performed in parallel in multiple cores. The similar idea has been adopted in the work from Karnagel et al. [47], to utilize the massively parallel computing power of GPU.

Handshake-Join (HSJoin): CellJoin essentially turns stream join into a scheduling and placement problem. Subsequently, it assumes that window partition and fetch have to be performed in global memory. Such assumption is later shown to be ineffective when the number of cores is large [87], and a new stream join technique called handshake join (i.e., HSJoin) was proposed. In contrast to CellJoin, handshake join adopts a data flow processing model (see Section 2.2). Specifically, both input streams notionally flow through the stream processing engine, in opposing directions. The two sliding windows are laid out side by side and predicate evaluations are performed along with the windows whenever two tuples encounter each other.

Low-Latency Handshake-Join (i.e., LLHSJoin): Despite its excellent scalability, the downside of HSJoin is that tuples may have to be queued for long periods of time before the match, resulting in high processing latency. In response, Roy et al. [76] propose a *low-latency* handshake-join (i.e., LLHSJoin) algorithm. The key idea is that, instead of sequentially forwarding each tuple through a pipeline of processing units, tuples are replicated and forwarded to all involved processing units (see the red dotted lines in Figure 2(c)) before the join computation is carried out by one processing unit (called home node).

SplitJoin: The state-of-the-art windowed join implementation called SplitJoin [65] parallelizes join processing in a top-down data flow model. Rather than forwarding tuples bidirectionally, as

in handshake join, SplitJoin broadcasts each newly arrived tuple (from either S or R), T to all processing units. In order to make sure each tuple is processed only once, T is retained in exactly one processing unit chosen in a round-robin manner. Although SplitJoin and HSJoin theoretically can achieve the same concurrency, the former has a much simpler processing logic and completely avoids central coordination in duelling with race conditions for tuples concurrently travelling between neighbouring processing cores as in HSJoin [65]. As a result, it shows a much higher throughput with actual implementation.

4.1.2 Windowed Stream Aggregation

Another computationally heavy windowed operator in stream applications is windowed stream aggregation, which summarizes the most recent information in a data stream. There are a number of different aggregation functions, such as "sum", "max", "sketch count". They differ in properties including *invertible*, *associative*, *commutative*, and *order-preserving*. A detailed description of these functions can be found in a recent survey [84].

Algorithm Overview. The trivial implementation is to perform the aggregation calculation from scratch for every arrived data. The complexity is hence $O(n)$, where n is the window size. Intuitively, efficiently leveraging previous calculation results for future calculation is the key to reduce computation complexity, which is often called as "incremental aggregation". However, the effectiveness of incremental aggregation relies on the property of the aggregation function. To name a few, when aggregation function is invertible (e.g., sum), we can simply update (i.e., increase) the aggregation results during tuple insert and evict with a complexity of $O(1)$. For faster answering median like function, which has to keep all the relevant inputs, instead of performing a sort on the window for each newly inserted tuple, we can maintain an *order statistics tree* as auxiliary data structure [39], which has $O(logn)$ worst-case complexity of its insert, delete and rank function. Similarly, the reactive aggregator (RA) [85] with $O(logn)$ average complexity only works for aggregation function with the associative property. Those algorithms are the best-performing approaches for a range of aggregation functions with different properties.

HW-Conscious Optimizations. There are a number of works on accelerating sliding windowed stream aggregation in a hardware-friendly manner. An early work from Mueller et al. [62] described implementation for a sliding windowed median operator on FPGAs. The algorithm skeleton adopted by the work is rather conventional: first sort elements within the sliding window and then compute the median. Compared to the $O(logn)$ complexity of using an order statistics tree as auxiliary data structure [39], Mueller's method has a theoretically much higher complexity due to the sorting step ($O(nlogn)$). Nevertheless, their key contribution lies in how the sorting and computing steps can be efficiently performed in FPGAs. Mueller's implementation [62] focuses on efficiently processing one sliding window without discussing handling subsequent sliding windows. The same author hence proposed to conduct multiple computations for each sliding window by instantiating multiple aggregation compute modules concurrently [63].

Recomputing from scratch for each sliding window is too costly even if conducted in parallel [63]. Hence, a technique called pane [55] was proposed and later verified on FPGAs [68] to address this issue. The key idea is to divide overlapping windows into disjoint panes, compute sub-aggregates over each pane, and "rolls up" the partial-aggregates to compute final results. Pane was later improved [50] and covers more cases (e.g., to support non-periodic windows [19]). However, these works are mostly theoretical and little work has been done to validate the effectiveness of these techniques on modern hardware, e.g., multicore processors.

Saber [49] is a relational stream processing system targeting heterogeneous machines equipped with CPUs and GPUs. To achieve high throughput, Saber also adopts incremental aggregate computations utilizing the commutative and associative property of some aggregation functions such as count, sum, average. Theodorakis et al. [88] recently studied the tradeoff between work and CPU efficient streaming window aggregation. To this end, they proposed an implementation that is both work- and CPU- efficient. Gong et al. [32] proposed an efficient and scalable accelerator based on FPGAs, called ShuntFlow, to support arbitrary window sizes for both reduce- and index-like sliding window aggregations. The key idea is partition aggregation with extremely large window sizes into sub-aggregations with smaller window sizes that can take more efficient use of FPGAs.

Contrary to previous works, Geethakumari et al. [29] argue that buffering all incoming raw data into a single window before processing is more efficient and is often the only option. This is

particularly true for small slide size and median-like operations because of the significant constant overhead of each sub-window. Instead of storing partial aggregated states as multiple overlapping windows [63], or panes [55], they simply store the input data into as a single window. Subsequently, the actual computation is started every time the window is full.

4.2 Deterministic Operator

The streaming event usually has an associated timestamp that indicates its temporal sequence. The deterministic operator often turns out to be a performance bottleneck as there is a fundamental conflict between data parallelism and determinism. The former seeks to improve the throughput of an operator by letting more than one thread operate on different events concurrently, possibly out-of-order. While determinism requires to strictly process each event following their timestamp sequence.

Algorithm Overview. Currently, there are three different approaches to ensuring determinism in stream processing. The first utilizes a buffer-based data structure [16] that buffers incoming tuples for a period of time before processing. The key idea is to keep the data as long as possible to avoid determinism violations. The second technique relies on punctuation [56], which is a special tuple in the event stream indicating the end of a sub-stream. Punctuations guarantee that tuples are processed in monotonically increasing time sequence across punctuations but not within two subsequent punctuations. The third approach is to use speculative techniques [77]. The main idea is to process tuples without any delay and recompute the results in case of determinism violation.

HW-Conscious Optimizations. Gulisano et al. [34] are among the first to provide a deterministic guarantee for high-performance stream join on multicores. The proposed algorithm, depicted in Figure 3(a), called *scalejoin* first merges all incoming tuples into one stream (through a data structure called *scalegate*) and then distributes them to processing threads (PTs) to perform join. The output also needs to be merged and sorted before exists the system. The use of scalegate makes this work fall into the category of buffer-based approach and the inherent limitation of higher processing latency. Scalejoin has been implemented in FPGA [51], and further improved in another recent work [75]. They both found that the proposed system outperforms the corresponding fully optimized parallel software-based solution running on a high-end 48-core multiprocessor

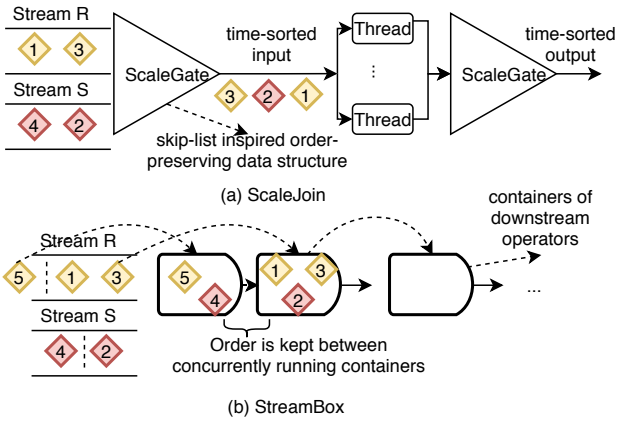

Figure 3: Multicore-friendly deterministic stream processing mechanisms.

platform.

Different from providing determinism in a specific operator (e.g., stream join [34]), StreamBox [61] provides determinism for all deterministic-sensitive operators from a system-level with punctuation based technique. Figure 3(b) illustrates the basic idea by taking the stream join operator as an example. Relying on a novel data structure called *cascading container* to track dependencies between epochs (a group of tuples delineated by punctuations), StreamBox is able to maintain the processing order among multiple concurrently executing containers that exploit the parallelism of modern multicore hardware.

A recent work from Kuralenok et al. [53] attempt to balance the conflict between deterministic and multicore parallelism with an optimistic approach falling in the third approach. The basic idea is to conduct process without any constraints, but apologize (i.e., sending amending signals) when determinism is violated. They show that the performance of the proposed deterministic approach depends on how often reorderings are observed during runtime. In the case when the order naturally preserved, there is almost no overhead. However, it leads to extra network traffic and computations when reorderings are frequent. To apply such an approach in practical use-cases, it is hence necessary to predict the probability of reordering, which could be interesting future work.

4.3 Remarks

There has been a rapid increase in the amount of data being generated and collected for analysis, making high-throughput processing essential to satisfy *Process and Respond Instantaneously (R8)* [82] requirement. However, achieving high-

throughput stream processing is challenging, especially when using those expensive operators. From the above discussion, it is clear that the key to accelerating windowed operators are mainly two folds. On one hand, we should minimize the operation complexity. There are two common approaches: 1) incremental computation algorithms [49], which maximizes reusing intermediate results and 2) rely on clever auxiliary data structures (e.g., indexing the contents of sliding window [94]) for reducing data (and/or instruction) accesses, especially cache misses. On the other hand, we should maximize operation concurrency. This requires us to distribute workloads among multiple threads and minimize synchronization overhead among threads [61]. Unfortunately, these optimization techniques are often at odds with each other. For example, incremental computation algorithm is complexity efficient, but difficult to parallelize due to inherent control-dependencies in the CPU instruction [88]. Another example is that maintaining index structures for partial computing results may help to reduce data access but it also brings maintenance overhead [57]. More investigation is required to better balance these conflicting aspects.

The system support of deterministic operator is relevant to both *Handle Stream Imperfections (R3)* and *Generate Predictable Outcomes (R4)* [82]. In real-time stream systems, where the input data is not stored, the infrastructure must make provision for handling data that arrive late or is delayed, missing or out-of-sequence. Correctness can be guaranteed only if time-ordered, deterministic processing is maintained throughout the entire processing pipeline. Despite the significant efforts in accelerating deterministic operators, existing DSPSs are still far from ideal in exploiting the potential of modern hardware.

5. COMMUNICATION OPTIMIZATION

We have discussed prior works on accelerating individual operators (i.e., the vertex of the DAG), and we now review the literature on improving the communication efficiency using modern hardware.

5.1 In-Core Communication

In contrast to the traditional "store & process" workflow, stream processing generally requires input events to be processed without storing them. By eliminating unnecessary I/O latency, modern DSPSs [1, 2] are able to achieve very low processing latency in the order of milliseconds. However, excessive communication among operators (i.e., *in-core communication* [18]) is still often a key obstacle in further improving the performance of DSPSs.

Kamburugamuve et al. [43] recently presented their findings on integrating Apache Heron [3] with Infiniband and Intel OmniPath. The results show that both can be utilized to improve the performance of distributed streaming applications. Nevertheless, much more optimization opportunities remain to be explored. For example, Kamburugamuve et al.[43] have evaluated Heron on Infiniband with channel semantics, and there is still remote direct memory access (RDMA) semantics available [71], which has shown to be very effective in other works [80, 96].

Data compression is a widely used approach to reducing communication overhead. Pekhimenko et al. [70] recently examined the potential of using data compression in stream processing. Interestingly, they found that data compression does not necessarily lead to a performance gain. Instead, improvement can be only achieved through a combination of hardware accelerator (i.e., GPU in their proposal) and new execution techniques (i.e., computation over compressed data).

Data grouping is necessary for stream applications. For instance, word-count requires the same word to be transmitted to the same `Counter` operator (see Section 2.1). Subsequently, all DSPSs need to implement data grouping operations, regardless of their processing model (i.e., continuous operator model or bulk synchronous model). Data grouping involves excessive memory accesses that rely on hash-based data structures [99, 95]. Zeuch et al. [97] analyzed the design space of DSPSs optimized for modern multicore processors. In particular, they show that a queue-less execution engine based on query compilation, which replaces communication between operators with function calls, is highly suitable for modern hardware. Since data grouping cannot be completely eliminated, they proposed a mechanism called "Upfront Partitioning with Late Merging", for efficient data grouping. Miao et al. [60] have exploited the possibility of accelerating data grouping using emerging 3D-stacked memories such as high-bandwidth memory (HBM). By designing the system in a way that addresses the limited capacity of HBM and HBM's need for sequential-access and high parallelism, the resulting system was able to achieve multi-times performance improvement over the baseline.

5.2 Out-of-Core Communication

Emerging stream applications often require the underlying DSPS to maintain large application state so as to support complex real-time analytics [89]. Representative example states required during stream processing include graph data structures [101] and transaction records [59]. Following previous works [18], we define the data flow between the stream operator and external storage as *out-of-core communication*.

The storage subsystem has undergone tremendous innovation in order to keep up with the ever-increasing performance demand. Non-Volatile Memory (NVM) has emerged as a promising hardware and brings many new opportunities and challenges. Fernando et al. [72] have recently explored efficient approaches to support analytical workload on NVM, where an NVM-aware storage layout for tables is presented based on a multi-dimensional clustering approach and a block-like structure to utilize the entire memory stack. As argued by the author, the storage structure designed on NVM may serve as the foundation for supporting features like transactional stream processing systems [5] in future.

Non-Volatile Memory Express (NVMe) based solid state devices (SSDs) are expected to deliver unprecedented performance in terms of latency and peak bandwidth. For example, the recently announced PCIe 4.0 based NVMe SSDs [6] is already capable of achieving a peak bandwidth of 4GB/s. Lee et al. [54] have recently investigated the performance limitations of current DSPSs on managing application states on SSDs and have shown that that query-aware optimization can significantly improve the performance of stateful stream processing on SSDs.

5.3 Remarks

Most DSPSs are designed with *Keep the Data Moving (R1)* [82] as the fundamental design principle, and hence aims to process input data "on-the-fly" without storing them. Subsequently, message passing is often a key component in current DSPSs. From the above discussion, we can see that, researchers have attempted to improve the communication efficiency by taking advantage of the latest advancement in network infrastructure, compression using hardware accelerator, and efficient algorithms exploiting new hardware characteristics. However, a model-guided approach to balance the tradeoff between computation and communication overhead is still in general missing in existing works.

The out-of-core communication is more related to *Integrate Stored and Streaming Data (R5)* [82] requirement. For many stream processing applications, comparing "present" with "past" is a common task. Thus, the system must also provide careful management of the stored state. However, we observe that only a few related studies attempt to improve out-of-core communication efficiency on modern hardware. There are still many open questions to be resolved, such as new storage formats, indexing techniques, for emerging hardware architectures and stream applications [5].

6. QUERY DEPLOYMENT

We now review prior works from a higher level of abstraction – the query/application dimension. In the following, we review existing works that utilize hardware-conscious features to assist stream processing in the following three categories: multicore CPUs, GPUs, and FPGAs.

6.1 Multicore Stream Processing

Language and Compiler. Multicore architectures have been ubiquitous. However, programming models and compiler techniques for employing multicore features are still lag behind hardware improvements. Kudlur et al. [52] were among the first to develop a compiler technique to map stream application to a multicore processor. By taking the Cell processor as an example, they study how to compile and run a stream application expressed in their proposed language `StreamIt`. The compiler works in two steps, 1) operator fission optimization (i.e., split one operator into multiple ones) and 2) assignment optimization (i.e., assign each operator to a core). The two-step mapping is formulated as an integer linear programming (ILP) problem and requires a commercial ILP solver. Noting its NP-Hardness, Farhad et al. [26] later presented an approximation algorithm to solve the mapping problem. Note that, the mapping problem from Kudlur et al. [52] considers only CPU loads, and ignores communications bandwidth. In response, Carpenter et al. [20] developed an algorithm that maps a streaming program onto a heterogeneous target, further taking communication into consideration. To utilize a SIMD-enabled multicore system, Hormati et al. [41] proposed to vectorize stream applications. Relying on high-level information, such as the relationship between operators, they were able to achieve better performance than general vectorization techniques. Agrawal et al. [11] proposed a cache-conscious scheduling algorithm for mapping stream

application on multicore processors. Particularly, they developed the theoretical lower bounds on cache misses when scheduling a streaming pipeline on multiple processors, and the upper bound of the proposed cache-based partitioning algorithm called *seg_cache*. They also experimentally found that scheduling solely based on the modelled cache effects can be often more effective than the conventional load-balancing (based on computation cost) approaches.

Multicore-aware DSPSs. Recently, there has been a fast growing interest in building multicore-friendly DSPSs. Instead of statically compiling a program as done in `StreamIt` [52, 26, 20], these DSPSs provide better elasticity for application execution. They also allow the usage of general purpose programming languages (e.g., Java, Scala) to express stream applications. Tang et al. [83] studied the data flow graph to explore the potential parallelism in a DSPS and proposed an *auto-pipelining* solution that can utilize multicore processors to improve the throughput of stream processing applications. For economic reasons, power efficiency has become more and more important in recent years, especially in HPC domains. Kanoun et al. [45] proposed a multicore scheme for stream processing that takes power constraint into consideration. Trill [22] is a single-node query processor for temporal or streaming data. Contrary to most distributed DSPSs (e.g., Storm, Flink) adopting continuous operator model, Trill runs the whole query only on the thread that feeds data to it – essentially a bulk-synchronous model. Such an approach has shown to be especially effective [97] when applications contain no synchronization barriers.

6.2 GPU-Enabled Stream Processing

GPUs are the most popular heterogeneous processors due to their high compute capacity and cost-effectiveness. However, due to their unique execution model, special designs are required to efficiently adapt stream processing to GPUs.

Single-GPU. Verner et al. [91] presented a general algorithm for processing data streams with real-time stream scheduling constraint on GPUs. This algorithm assigns data streams to CPUs and GPUs based on their incoming rates. It tries to provide an assignment that can satisfy different requirements from various data streams. Zhang et al. [98] developed a holistic approach to build DSPSs using GPUs. They design a latency-driven GPU-based framework, which mainly focuses on real-time stream processing. Due to the limited memory capacity of GPUs, the window size of the stream operator plays an important role in system performance. Pinnecke et al. [73] studied the influence of window size, and proposed a partitioning method for splitting large windows into different batches, considering both time and space efficiency. SABER [49] is a window-based hybrid stream processing framework using GPUs. SABER focuses on stream SQL queries; during execution and support execution involving both CPUs and GPUs.

Multi-GPUs. Multi-GPUs systems provide tremendous computation capacity, but also pose challenges like how to partition or scheduling among different GPUs. Verner et al. [92] extend their method [91] to a single node with multiple GPUs. A scheduler manipulates stream placement and guarantees that the requirements among different streams can be met. GStream [102] is the first data streaming framework for GPU clusters. GStream supports stream processing applications in the form of a C++ library; it uses MPI to implement the data communication between different nodes, and uses CUDA to conduct stream operations on GPUs. In addition, GStream includes a language abstraction for users to describe different applications. Alghabi et al. [14] first introduced the concept of stateful stream data processing on a node with multiple GPUs. This work describes processing graph-like arrangements of different processing modules. Nguyen et al. [67] considered the scalability with the number of GPUs on a single node, and developed a GPU performance model for stream workload partitioning in multi-GPU platforms with high scalability. Chen et al. [24] proposed G-Storm, which enables Storm [2] to utilize GPUs and can be applied to various applications that Storm has already supported.

6.3 FPGA-Enabled Stream Processing

FPGAs are programmable integrated circuits whose hardware interconnections can be configured by users. Due to their low latency, high energy efficiency, and low hardware engineering cost, FPGAs have been explored in various application scenarios, including stream processing.

Hagiescu et al. [35] first elaborated challenges to implementing stream processing on FPGAs, and proposed algorithms that optimizes processing throughput and latency for FPGAs. Mueller et al. [63] provided *Glacier*, which is an FPGA-based query engine that can process queries from streaming networks. The operations in Glacier include selection, aggregation, grouping,

and windowing. Experiments show that using FPGAs help achieve much better performance than using conventional CPUs. A common limitation of an FPGA-based system is its intensive synthesis process, which takes significant time to compile the application into hardware designs for FPGAs. This makes FPGA-based systems inflexible in adapting to query changes. In response, Najafi et al. [64] demonstrated Flexible Query Processor (FQP), an online reconfigurable event stream query processor that can accept new queries without disrupting other queries under execution.

6.4 Remarks

Existing systems usually involve heterogeneous processors along with CPUs. Such heterogeneity opens up both new opportunities and poses challenges for scaling stream processing. From the above discussion, it is clear that both GPUs and FPGAs have been successfully applied for scaling-up stream processing. *Partition and Scale Applications Automatically (R7)* [82] requires a DSPS to be able to elastically scale up and down in order to process input streams with varying characteristics. However, based on the above analysis, little work has considered scaling *down* the processing efficiently (and easily scale up later) in a hardware-aware manner. A potential direction is adopting server-less computing paradigm [17] into DSPSs. However, how to efficiently manage the partial computing state in GPUs or FPGAs still remain unclear. FPGAs have low latency and are hardware configurable. Hence, they are suitable for special application scenarios, such as a streaming network. However, to the best of our knowledge, there has been no prior works exploring the potential of multiple FPGAs for stream processing.

7. CONCLUSION

The proliferation of high-rate data sources has accelerated the development of next-generation performance-critical DSPSs. For example, the new 5G network promises blazing speeds, massive throughput capability and ultra-low latencies [7], thus bringing the higher potential for performance critical stream applications. In this paper, we have discussed relevant literature from the field of hardware-conscious DSPSs, which aim to utilize modern hardware capabilities for accelerating stream processing. Those works have significantly improved DSPSs to better satisfy the design requirements raised by Stonebraker et al. [82]. However, there are two requirements including *Query using SQL on Streams (R2)* and *Guarantee Data Safety and Availability (R6)*, that is overlooked by most existing HW-conscious DSPSs. In particular, how to design HW-aware SQL statements for DSPSs, and how best to guarantee data safety and system availability when adopting modern hardware, such as non-volatile memory for local backup and high-speed network for remote backup, remain as open questions.

7.1 Future Directions

Efficient Stream Aggregation. From our discussion in Section 4, it is clear that parallelizing stream joins to accelerate computation on multi-core and many-core architectures have received significant attention. Overall, new algorithms, designs and architectures were proposed and tested on new hardware. In contrast, studies on the potential of modern hardware in accelerating stream aggregation are still mostly theoretical. Hence, it might be interesting to see how the state-of-the-art stream aggregation algorithm (e.g., utilizing order statistics tree [39] for stream median) can be efficiently implemented in the multicore processor, GPU, or FPGAs.

Scale-up and -out Stream Processing. As emphasized by Gibbons [30], scaling both out and up is crucial to effectively improve the system performance. In-situ analytics enable data processing at the point of data origin, thus reducing the data movements across networks; Powerful hardware infrastructure provides an opportunity to improve processing performance within a single node. To this end, many recent works have exploited the potential of high-performance stream processing on single node [49, 61, 97]. However, the important question of how best to use powerful local nodes in the context of large distributed computation setting still remains unclear.

Stream Processing Processor. Previous work [28, 78] have found that the Cell processor is particularly useful for accelerating stream processing. Specifically, its programmable local storage, low latency and high bandwidth access to main memory are ideal for parallelizing stream processing [78]. With the wide adopting of stream processing today, it may be a good time to revisit the design of special hardware/processor for DSPSs. The requirement for such processors include *low-latency*, *low power consumption* and *high bandwidth*. Further, components like complex control logic may be sacrificed as stream processing logic is usually predefined and fixed.

8. REFERENCES

[1] Apache Flink. https://flink.apache.org/.

[2] Apache Storm. http://storm.apache.org/.

[3] Heron, url: https://twitter.github.io/heron/.

[4] SGI UVTM 300H System Specifications. https://www.sgi.com/pdfs/4559.pdf.

[5] Transactional stream processing on non-volatile memory.

[6] Corsair force series nvme ssd. https://www.anandtech.com/show/14416/corsair-announces-mp600-nvme-ssd-with-pcie-40, 2019.

[7] Ericsson mobility report november 2018, https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-november-2018.pdf, 2019.

[8] Number of connected iot devices will surge to 125 billion by 2030, ihs markit says. https://technology.ihs.com/596542/number-of-connected-iot-devices-will-surge-to-125-billion-by-2030-ihs-markit-says, 2019.

[9] D. Abadi and et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2), 2003.

[10] D. J. Abadi and et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[11] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo. Cache-conscious scheduling of streaming applications. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 236–245, New York, NY, USA, 2012. ACM.

[12] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013.

[13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[14] F. Alghabi, U. Schipper, and A. Kolb. A scalable software framework for stateful stream data processing on multiple gpus and applications. In *GPU Computing and Applications*, pages 99–118. Springer, 2015.

[15] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[16] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, Sept. 2004.

[17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. Springer Singapore, Singapore, 2017.

[18] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, Aug. 2017.

[19] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. Cutty: Aggregate sharing for user-defined windows. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 1201–1210, New York, NY, USA, 2016. ACM.

[20] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 57–66, New York, NY, USA, 2009. ACM.

[21] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.

[22] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. Platt, J. Terwilliger, J. Wernsing, and R. DeLine. Trill: A high-performance incremental query processor for diverse analytics. VLDB - Very

Large Data Bases, August 2015.

[23] S. Chandrasekaran and et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[24] Z. Chen, J. Xu, J. Tang, K. Kwiat, and C. Kamhoua. G-Storm: GPU-enabled high-throughput online data processing in Storm. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 307–312. IEEE, 2015.

[25] J. A. Colmenares, R. Dorrigiv, and D. G. Waddington. Ingestion, indexing and retrieval of high-velocity multidimensional sensor data on a single node. volume abs/1707.00825, 2017.

[26] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation: Mapping stream programs onto multicore architectures. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 357–368, New York, NY, USA, 2011. ACM.

[27] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[28] B. Gedik, R. R. Bordawekar, and P. S. Yu. Celljoin: A parallel stream join operator for the cell processor. *The VLDB Journal*, 18(2):501–519, Apr. 2009.

[29] P. R. Geethakumari, V. Gulisano, B. J. Svensson, P. Trancoso, and I. Sourdis. Single window stream aggregation using reconfigurable hardware. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 112–119, Dec 2017.

[30] P. B. Gibbons. Big data: Scale down, scale up, scale out. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 3–3, May 2015.

[31] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 500–511. VLDB Endowment, 2003.

[32] S. Gong, J. Li, W. Lu, G. Yan, and X. Li.

[33] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGPLAN Not.*, 41(11):151–162, Oct. 2006.

[34] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 144–153, Oct 2015.

[35] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. Rabbah. A computing origami: folding streams in FPGAs. In *2009 46th ACM/IEEE Design Automation Conference*, pages 282–287. IEEE, 2009.

[36] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream window join: tracking moving objects in sensor-network databases. In *15th International Conference on Scientific and Statistical Database Management, 2003.*, pages 75–84, July 2003.

[37] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, Aug. 2013.

[38] M. Hirzel and et al. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 2014.

[39] M. Hirzel, R. Rabbah, P. Suter, O. Tardieu, and M. Vaziri. Spreadsheets for stream processing with unbounded windows and partitions. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 49–60, New York, NY, USA, 2016. ACM.

[40] M. Hirzel, S. Schneider, and K. Tangwongsan. Sliding-window aggregation algorithms: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 11–14, New York, NY, USA, 2017. ACM.

[41] A. H. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Macross: Macro-simdization of streaming applications. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 285–296, New York, NY, USA, 2010. ACM.

[42] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages

431–442, New York, NY, USA, 2006. ACM.

[43] S. Kamburugamuve, K. Ramasamy, M. Swany, and G. Fox. Low latency stream processing: Apache heron with infiniband & intel omni-path. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, UCC '17, pages 101–110, New York, NY, USA, 2017. ACM.

[44] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003.

[45] K. Kanoun, M. Ruggiero, D. Atienza, and M. Van Der Schaar. Low power and scalable many-core architecture for big-data stream computing. In *2014 IEEE Computer Society Annual Symposium on VLSI*, pages 468–473. IEEE, 2014.

[46] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. The hells-join: A heterogeneous stream join for extremely large windows. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pages 2:1–2:7, New York, NY, USA, 2013. ACM.

[47] T. Karnagel, B. Schlegel, D. Habich, and W. Lehner. Stream join processing on heterogeneous processors. In *BTW Workshops*, 2013.

[48] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. *Proc. VLDB Endow.*, 10(11):1286–1297, Aug. 2017.

[49] A. Koliousis and et al. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*, 2016.

[50] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 623–634, New York, NY, USA, 2006. ACM.

[51] C. Kritikakis, G. Chrysos, A. Dollas, and D. N. Pnevmatikatos. An fpga-based high-throughput stream join architecture. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016.

[52] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, June 2008.

[53] I. E. Kuralenok, N. Marshalkin, A. Trofimov, and B. Novikov. An optimistic approach to handle out-of-order events within analytical stream processing. In *CEUR Workshop Proceedings*, volume 2135, pages 22–29. RWTH Aahen University, 2018.

[54] G. Lee, J. Eo, J. Seo, T. Um, and B.-G. Chun. High-performance stateful stream processing on solid-state drives. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, pages 9:1–9:7, New York, NY, USA, 2018. ACM.

[55] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, Mar. 2005.

[56] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: A new architecture for high-performance stream systems. *Proc. VLDB Endow.*, 1(1):274–288, Aug. 2008.

[57] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 811–825, New York, NY, USA, 2015. ACM.

[58] N. Marz. Trident API Overview. `github.com/nathanmarz/storm/wiki/Trident-APIOverview`.

[59] J. Meehan and et al. S-store: streaming meets transaction processing. *Proc. VLDB Endow. 2015*.

[60] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox-hbm: Stream analytics on high bandwidth hybrid memory. *arXiv preprint arXiv:1901.01328*, 2019.

[61] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox: Modern stream processing on a multicore machine. In *ATC*, 2017.

[62] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[63] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for FPGAs. *Proceedings of the VLDB Endowment*, 2(1):229–240, 2009.

[64] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on FPGAs. *Proceedings of the VLDB Endowment*, 6(12):1310–1313, 2013.

[65] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Splitjoin: A scalable, low-latency stream join architecture with adjustable ordering

precision. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 493–505, Denver, CO, 2016. USENIX Association.

[66] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDMW*, 2010.

[67] D. Nguyen and J. Lee. Communication-aware mapping of stream graphs for multi-gpu platforms. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 94–104, New York, NY, USA, 2016. ACM.

[68] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga. An efficient and scalable implementation of sliding-window aggregate operator on fpga. In *2013 First International Symposium on Computing and Networking - Across Practical Development and Theoretical Research (CANDAR)*, pages 112–121, Los Alamitos, CA, USA, dec 2013. IEEE Computer Society.

[69] F. Pan and H. Jacobsen. Panjoin: A partition-based adaptive stream join. *CoRR*, abs/1811.05065, 2018.

[70] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou. Tersecades: Efficient data compression in stream processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 307–320, Boston, MA, 2018. USENIX Association.

[71] G. F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[72] G. Philipp, B. Stephan, and S. Kai-Uwe. An nvm-aware storage layout for analytical workloads. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 110–115, April 2018.

[73] M. Pinnecke, D. Broneske, and G. Saake. Toward GPU Accelerated Data Stream Processing. In *GvD*, pages 78–83, 2015.

[74] C. Pohl. Stream processing on high-bandwidth memory. In *Grundlagen von Datenbanken*, pages 41–46, 2018.

[75] C. Rousopoulos, E. Karandeinos, G. Chrysos, A. Dollas, and D. N. Pnevmatikatos. A generic high throughput architecture for stream processing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–5, Sep. 2017.

[76] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *Proc. VLDB Endow.*, 7(9):709–720, May 2014.

[77] E. Ryvkina, A. S. Maskey, M. Cherniack, and S. Zdonik. Revision processing in a stream processing engine: A high-level design. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 141–141, April 2006.

[78] S. Schneider, H. Andrade, B. Gedik, K.-L. Wu, and D. S. Nikolopoulos. Evaluation of streaming aggregation on parallel hardware architectures. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 248–257, New York, NY, USA, 2010. ACM.

[79] A. Shahvarani and H.-A. Jacobsen. Parallel index-based stream join on a multicore cpu. `https://arxiv.org/pdf/1903.00452.pdf`, 2019.

[80] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 317–332, Savannah, GA, 2016. USENIX Association.

[81] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 324–335. VLDB Endowment, 2004.

[82] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, Dec. 2005.

[83] Y. Tang and B. Gedik. Autopipelining for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2344–2354, 2013.

[84] K. Tangwongsan, M. Hirzel, and S. Schneider. *Sliding-Window Aggregation Algorithms*, pages 1–6. Springer International Publishing, Cham, 2018.

[85] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *Proc. VLDB Endow.*, 8(7):702–713, Feb. 2015.

[86] W. B. Teeuw and H. M. Blanken. Control versus data flow in parallel database machines. *IEEE Transactions on Parallel*

and Distributed Systems, 4(11):1265–1279, Nov 1993.

[87] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 625–636, New York, NY, USA, 2011. ACM.

[88] G. Theodorakis, A. Koliousis, P. Pietzuch, and P. Holger. Hammer slide: work-and cpu-efficient streaming window aggregation. 2018.

[89] Q.-C. To, J. Soto, and V. Markl. A survey of state management in big data processing systems. *The VLDB Journal*, 27(6):847–872, Dec. 2018.

[90] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM.

[91] U. Verner, A. Schuster, and M. Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 120–129, New York, NY, USA, 2011. ACM.

[92] U. Verner, A. Schuster, M. Silberstein, and A. Mendelson. Scheduling processing of real-time data streams on heterogeneous multi-gpu systems. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 8:1–8:12, New York, NY, USA, 2012. ACM.

[93] J. Xie and J. Yang. *A Survey of Join Processing in Data Streams*, pages 209–236. Springer US, Boston, MA, 2007.

[94] Y. Ya-xin, Y. Xing-hua, Y. Ge, and W. Shan-shan. An indexed non-equijoin algorithm based on sliding windows over data streams. *Wuhan University Journal of Natural Sciences*, 11(1):294–298, Jan 2006.

[95] M. Zaharia and et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, 2013.

[96] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, Feb. 2017.

[97] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment*, 12(5):516–530, 2019.

[98] K. Zhang, J. Hu, and B. Hua. A holistic approach to build real-time stream processing system with gpu. *Journal of Parallel and Distributed Computing*, 83:44 – 57, 2015.

[99] 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 *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 659–670. IEEE, 2017.

[100] S. Zhang, H. T. Vo, D. Dahlmeier, and B. He. Multi-query optimization for complex event processing in sap esp. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1213–1224. IEEE, 2017.

[101] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 614–630, New York, NY, USA, 2017. ACM.

[102] Y. Zhang and F. Mueller. Gstream: A general-purpose data streaming framework on gpu clusters. In *2011 International Conference on Parallel Processing*, pages 245–254, Sep. 2011.