Benchmarking Stream Join Algorithms on GPUs: A Framework and Its Application to the State of the Art

Dwi P. A. Nugroho
Technische Universität Berlin
d.nugroho@tu-berlin.de

Philipp M. Grulich
Technische Universität Berlin
grulich@tu-berlin.de

Steffen Zeuch
Technische Universität Berlin
steffen.zeuch@tu-berlin.de

Clemens Lutz
Technische Universität Berlin
clemens.lutz@tu-berlin.de

Stefano Bortoli
Huawei German Research Center
stefano.bortoli@huawei.com

Volker Markl
Technische Universität Berlin,
DFKI GmbH
volker.markl@tu-berlin.de

ABSTRACT

Joining streams is an essential and resource-demanding operation in stream processing engines (SPEs). Recent works have shown significant performance benefits by offloading stream-join processing to hardware accelerators, including GPUs. As a result, a wide variety of GPU-accelerated stream join algorithms (SJAs) have emerged. However, existing works evaluate the proposed GPU-accelerated SJAs only in isolation, on different hardware, and not using a common workload. As a result, it is difficult to compare different SJAs and select the best-suited SJA for a particular situation.

In this paper, we shed light on the performance characteristics of GPU-accelerated SJAs. To this end, we explore the configuration parameter space of SJAs and investigate the impact of each parameter. We evaluate the performance of well-known SJAs under multiple configurations of the underlying join algorithm, the parallelization strategy, the algorithm progressiveness, and the GPU type. Our results show that each variant of SJA has its strengths and weaknesses and that ill-suited configurations of parameters lead to up to two orders of magnitude difference in throughput. Based on the results, we developed a guideline for selecting SJA variants for different circumstances.

1 INTRODUCTION

Modern data analytic workloads, such as traffic analysis [8], system monitoring [12], online advertisement [1], and managing user engagement in online games [14] leverage event streams originating from multiple sources to produce rich analyses. To this end, the underlying analytic pipeline needs to join these event streams before processing them further. State-of-the-art stream processing engine (SPE) provide the stream join operator to accomplish such operation [4] [34] [29]. However, their scale-out solution does not exploit the underlying hardware efficiently in facing the increasing rate of data streams [35].

Recently, the research community has investigated different methods to scale stream join computation using Graphic Processing Units (GPUs) [15, 18, 19, 23, 38]. However, existing works evaluate their algorithm in isolation. Thus, it is difficult to compare the results of different papers. In turn, it is unclear which algorithm is the best under which circumstances.

In the following, we identify three issues that hinder us from comparing SJAs proposed in previous works.

Table 1: Different stream join workloads in related works.

Related Work	Join Predicate	Window Type	Window Measure
HELLSJoin [15]	Band Join	Sliding	Time-based
SABER [18]	Theta Join	Tumbling	Count-based
EPHSA [19]	Theta Join	Sliding	Count-based
FineStream [38]	Theta Join	Tumbling	Count-based
EcoJoin [23]	Equi Join	Interval	Time-based

Table 2: Evaluation parameters used in related works.

Related Work	Window Size Scaling	Batch Size Scaling	Comparison with CPUs
HELLSJoin [15]	Yes	No	Yes
SABER [18]	No	Yes	No
EPHSA [19]	Yes	No	Yes
FineStream [38]	No	No	No
EcoJoin [23]	No	Yes	Yes

Lack of a comparable workload. First, existing works use different stream join workloads, as shown in Table 1. Existing works evaluate workloads with different combinations of join predicates, window types, and window measures. In terms of join predicate, existing works use band-joins, theta joins, or equi-joins. In terms of window type, they use either a sliding window, tumbling window, or interval window. Existing works also evaluate different window measures, including time- and count-based windows. Different workloads have different semantics, and thus their performance comparison is not meaningful.

Unsystematic evaluation of the configuration parameter space of SJA. Second, existing works evaluate different parameters to demonstrate the performance of their algorithm. In Table 2, we list the experiment parameters evaluated in existing work. Only HELLSJoin [15] and EPHSA [19] investigate varying window sizes. Concerning batch size, only SABER [18] and EcoJoin [23] explore its impact on the performance of their algorithm. Finally, only HELLSJoin [15] and EPHSA [19] are concerned with the performance comparison of their algorithm to a single-threaded implementation of stream join on CPUs. Thus, each evaluation only shows us a snapshot rather than an overarching view of the performance characteristics of SJAs. Furthermore, existing works also do not consider other parameters that could affect the performance of a stream join implementation, such as

© 2024 Copyright held by the owner/author(s). Published in Proceedings of the 27th International Conference on Extending Database Technology (EDBT), 25th March-28th March, 2024, ISBN 978-3-89318-094-3 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Series ISSN: 2367-2005

10.48786/edbt.2024.17

Table 3: Metrics measured in related works.

Related Work	Measured Metric
HELLSJoin [15]	Latency
SABER [18]	Throughput and Latency
EPHSA [19]	Relative Speedup to CPU-based Approaches
Finestream [38]	Latency, Price-Throughput Ratio
EcoJoin [23]	Throughput, Latency, Energy Consumption

the distribution of the join keys, the number of distinct keys, the parallelization strategy, and the progressiveness of the algorithm. Hence, the configuration parameter space and the robustness of the proposed GPU-accelerated SJAs are unclear.

Missing or incomparable performance metric. Third, existing works in GPU-accelerated SJAs report varying performance metrics, as listed in Table 3. These metrics include throughput, latency, relative speed-up, price-throughput ratio, and energy consumption. However, existing works use different combinations of these metrics. Thus, it is difficult to compare the reported performance of the proposed SJAs

Our Contribution. In this paper, we shed light on performance-impacting factors of GPU-accelerated SJAs and enable comparison of different variants of SJAs through a structured experiment and analysis. In particular, we make the following contribution.

(1) We describe the configuration parameter space of GPU-accelerated SJAs, which includes the choice of the underlying join algorithm, the progressiveness mode of an SJA, the parallelization strategy, and the type of GPUs to execute an SJA.
(2) To reduce the complexity of exploring the configuration parameter space, we develop a general and extensible benchmarking framework for SJAs. We make our benchmarking framework available in our repository to allow further extension.
(3) We reveal the performance implications of each parameter and examine their performance-limiting factors.
(4) We create a clear guideline for choosing the best SJA for a given use case.

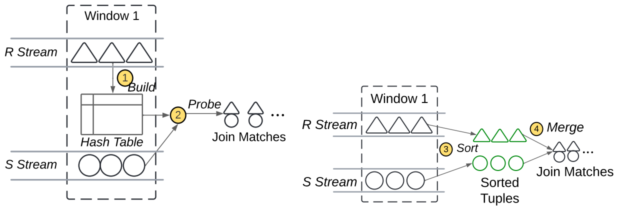
We organize the remainder of this paper as follows. We first recap the fundamentals of stream joins and query processing on GPUs in Section 2. In Section 3, we describe our experimental methodology and approach for conducting the experiments and analysis. We present the results of our experiments and give an interpretation in Section 4. In Section 5, we present a holistic discussion of all experimental results and provide a recommendation on how to choose a variant of SJA for a certain circumstance. We list out related works in Section 6. Finally, we provide an overall conclusion of our paper and potential future work in Section 7.

2 BACKGROUND

In this section, we introduce the semantics and the design space of stream join algorithms (Sec. 2.1) and provide an overview of data processing on GPUs (Sec. 2.2).

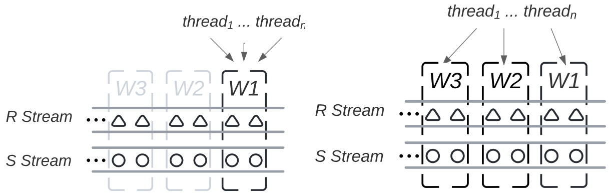
2.1 Stream Join

A stream join operator processes potentially unbounded streams. A stream $\bar{s}$ is a sequence of tuples $(t_1, t_2, ...)$, where each tuple is structured according to a given schema T [5]. To find join matches, an SJA first discretizes the continuous stream $\bar{s}$ into finite windows. A window $w_i = \bar{s}([b_i, e_i])$ is a substream of $\bar{s}$, with the tuples b_i and e_i located at the beginning and the end of w_i in stream $\bar{s}$ [13, 28]. Following Shahvarani et al. [27], we denote a stream join operation as $w_i^l \bowtie_\theta w_j^r$, where w_i^l and w_j^r are windows in two streams $\bar{s}_l$ and $\bar{s}_r$, and θ is the join predicate. An SJA emits all pairs $(p_{\hat{i}}^l, p_{\hat{j}}^r)$ as join results such that $p_{\hat{i}}^l \in w_i^l$ and $p_{\hat{j}}^r \in w_j^r$ and the predicate θ is satisfied. We can characterize an SJA by its underlying join algorithm, parallelization strategy, and progressiveness mode.



(a) Hash Join. (b) Sort-Merge Join.

Figure 1: Two underlying algorithms to execute stream join.



(a) Intra-window Join. (b) Inter-window Join.

Figure 2: Two strategies for parallelizing stream joins.

2.1.1 *Join Algorithm.* The underlying join algorithm defines the logic to find join matches in two streams. To this end, existing SJAs discretize the join evaluation of unbounded streams in windows. Thus, the join state and the join evaluation of a window are independent of other windows. In the case of Streaming Hash Join (SHJ) (Figure 1a), the algorithm first builds a hash table using tuples from one stream source in each window[23]. Subsequently, it probes the hash table using tuples from the other stream source in the same window to find join matches. In the case of Streaming Sort-Merge Join (SSMJ) (Figure 1b), the algorithm first sort the tuples from both stream sources in each window and then find join matches by scanning the sorted tuple from both stream sources in the same window [7, 39].

2.1.2 *Parallelization Strategy.* An SPE can execute stream join workloads using three different parallelization strategies, i.e., intra-window parallelization, inter-window parallelization, and hybrid window parallelization. With the intra-window parallelization strategy (Figure 2a), an SPE employs multiple threads to evaluate a single window and execute subsequent windows sequentially. This strategy is common in existing stream join algorithms [15, 18, 19, 23, 39]. Alternatively, with the inter-window parallelization strategy (Figure 2b), an SPE utilizes multiple threads to evaluate joins on different windows at a time. Each thread is responsible for the execution of a window. Third, an SPE can combine intra and inter-window parallelization strategies in a hybrid strategy, i.e., executing multiple windows simultaneously,

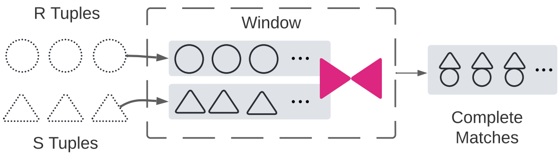


Figure 3: Lazy progressiveness mode.

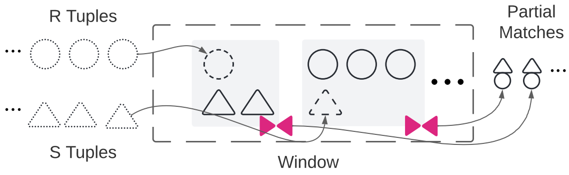


Figure 4: Eager progressiveness mode.

each utilizing multiple threads. The choice of which strategy to choose may impact the performance of the SJA.

2.1.3 Algorithm Progressiveness. Depending on how SJAs make progress in delivering stream join results, we can characterize SJAs to two different progressiveness modes [39]. First, in the *lazy* mode (Figure 3), an SJA buffers incoming tuple until it reaches a trigger to evaluate the join operation, e.g., when the window is closed in the case of a tumbling window (i.e., a consecutive and non-overlapping windows of equal size [31]). Thus, it results in a latency between the tuples's arrival and the join evaluation. However, this characteristic allows SJAs to evaluate the join on a larger granularity.

Second, in the *eager* mode (Figure 4), an SJA starts producing results as soon as a tuple (or a batch of tuples) arrives at the system. To this end, SJAs with eager progressiveness mode perform symmetric operations on both sides to find join matches [28, 32]. As a result, the eager progressiveness mode minimizes the latency between arrivals and the join execution of the tuples. This characteristic is advantageous for cases such as interactive applications [20], data visualization [7], and when operating on unreliable infrastructures [24]. However, the eager mode also triggers more function invocations to find join matches.

2.2 Query Processing on GPUs

A data processing system has to address challenges that arise when utilizing GPUs, including a distinct processing model, interconnect bottleneck, and large configuration space.

A Different Hardware Architecture. GPUs have a different architecture and design trade-off compared to CPUs. CPUs leverage complex microarchitecture features (e.g., branch prediction, speculative execution, out-of-order execution) to optimize the latency of few, usually compute-intensive threads [25]. In contrast, GPUs provide high throughput by exploiting explicit data parallelism. To this end, GPU architectures feature a large number of simple processing cores compared to CPUs. Latency hiding in GPUs is enabled by oversubscriptions of threads, i.e., allowing a single streaming multiprocessor (SM) to manage the execution of many threads. The capability to store states of threads in (relatively) large SM registers allows the GPU to switch between threads with low overhead compared to CPUs [25].

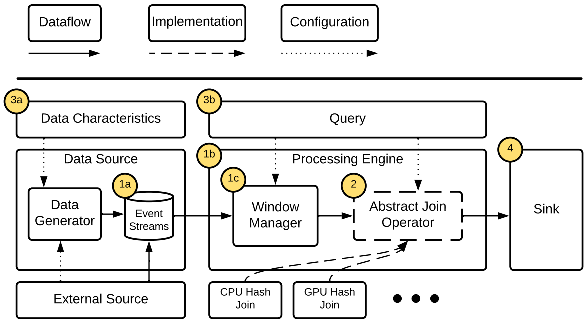


Figure 5: A benchmarking framework for SJAs.

Interconnect Bottleneck. Data processing on the GPUs involves transferring the input data to the GPU and transferring the output back from the GPU through an interconnect. In the case of PCI-E 3.0 interconnect, the bandwidth is smaller than the system's main memory and GPU memory bandwidth. Thus, the data transfer through interconnects with small bandwidth becomes one of the major bottlenecks in GPU-accelerated data processing systems [3, 21, 22].

Furthermore, GPUs have smaller memory capacity compared to CPUs [3, 21, 25]. A single high-end GPU (e.g., Nvidia V100) has only up to 32GB of memory. Processing data larger than GPU memory requires spilling the data to the host memory. This process triggers more data movement between the GPU and system's main memory [21].

Large Configuration Space. GPUs have multiple parameters that must be configured appropriately before executions, e.g., grid sizes, block sizes, and amount of shared memory per block. The configuration may vary depending on the type of each executed workload. Furthermore, the best configuration for a type of GPU hardware may not translate well to other types of GPUs, e.g., between dedicated GPUs (dGPU) and integrated GPUs (iGPUs) [26].

3 METHODOLOGY

In this section, we describe our benchmarking framework (Sec. 3.1), the performance metrics (Sec. 3.2), default parameters (Sec. 3.3), and the hardware setup (Sec 3.4).

3.1 Benchmarking Framework

As no common framework exists for evaluating different SJAs, we develop an extensible stream join benchmarking framework. We illustrate our benchmarking framework in Figure 5. We aim to provide a benchmarking framework that delivers accurate, precise, reproducible results and is extensible to future SJAs. Thus, we design our benchmarking framework to fulfill the following goals.

Low Overhead. Our benchmarking framework separates the data generator **1a** from the processing engine **1b** to isolate the measurement of the performance of the evaluated SJA. Our data generator initializes a buffer of event streams and stores them in the main memory. Thus, it does not incur memory allocation and data generation overhead during executions of SJAs. To produce a streaming workload, our data generator continuously pushes a batch of pre-generated event streams to the window manager in the processing engine. The window manager 1c is responsible for assigning the incoming tuples to a window. We store the assignment of tuples to windows in a queue for bookkeeping. The window manager handles the incoming tuple in two different ways. First, for SJAs that do not require buffering, e.g., SJAs with the eager progressiveness mode (see Section 4.3), the window manager updates the assignment of windows and forwards incoming tuples directly to the SJA. Second, for SJAs that require buffering, the window manager uses two circular buffers, i.e., one for each join side, to store the incoming tuples temporarily. In this case, the assignment contains the index of the circular buffer. SJAs utilize these indexes to read event streams to join from the circular buffer. When a window is expired, the window manager pops the indexes of circular buffers belonging to the window from the queue, i.e., allowing the next incoming tuples to write in those buffers.

Reusable Components. We design our benchmarking framework to be extensible and allow for evaluations of different SJAs on common ground. To this end, we decouple the implementation of SJAs from the processing engine using an abstract interface 2. Each concrete SJA implements abstract methods to produce join results from the given input. This abstraction makes our benchmarking framework independent of the implementation of the SJA, e.g., with regard to the target hardware.

In Figure 6 we show the abstract interface of SJAs in our benchmarking framework. In particular, we emphasize two important aspects. First, our benchmarking framework features a generic *Base Window Manager* that provides two interfaces to receive *Left* and *Right* tuples from the stream during runtime. With this interface, we can implement different kinds of windowing logic. Our Window Manager also contains a pointer to an implementation of an SJA, from which it can invoke the join evaluation. Second, our generic SJA interface provides an *execute()* method that accepts arrays of *Left* and *Right* tuples (i.e., collected by the Window Manager). In the case of a GPU implementation, it copies these arrays to the GPU memory and performs the join operation. This interface allows us to implement different operations required by an SJA to produce join results, such as managing join states. At the end of the operation, it returns the number of join matches and the actual result tuples (i.e., containing the join keys and the payload from both sides).

We publish our benchmarking framework in our code repository[1]. Thus, allowing for further extension of the benchmark by evaluating custom SJAs, custom windowing logic, or experimenting with different execution environments.

Configurable Query and Data Characteristics. To investigate the behavior of each SJA under different circumstances, we design the query and data characteristics used in the benchmark to be configurable. To this end, we parameterize the stream characteristics 3a, including the number of distinct keys and the key distribution for the generated data. In terms of query characteristics 3b, we design our window manager to accept different window sizes.

Result Verification. Each join implementation has access to the sink 4. The sink contains a match counter and an output buffer that can be modified by an SJA implementation. Thus, we can verify the correctness of an SJA by comparing the produced tuples with the expected result.

Table 4: Default data and query characteristics.

Parameter	Domain	Default Value
Number of Distinct Keys	Integer	1000
Zipf Exponent	Float	0
Batch Size	(in KB) Integer	128 KB
Window Size	(in MB) Integer	1 MB

Table 5: Default parameters for SJA configuration.

Parameter	Domain	Default Value
Join Algorithm	{Nested Loop Join, Hash Join, Sort-Merge Join}	Hash Join
Parallelization Strategy	{Inter-Window, Intra-Window, Hybrid}	Inter-Window
Progressiveness	{Lazy, Eager}	Lazy
Result Materialization	{No Materialization, Count-Kernel,Atomic, Estimated Selectivity}	No Materialization

3.2 Performance Metrics

In this paper, we consider four metrics for comparing SJAs. First, we measure the maximum sustainable throughput of each join algorithm. Karimov et al. [14] define the maximum sustainable throughput as the highest load the system can handle without inducing backpressure. Second, we measure the average window processing time, i.e., the time an algorithm requires to fully compute the join matches on a single window. Third, we measure the progressiveness of SJAs. To this end, we measure the cumulative percentage of produced tuples over time. Fourth, we also measure the energy efficiency of the SJAs.

3.3 Default Parameters

Based on our exploratory evaluations, we define several default parameters used in our experiments as we show in Table 4 and Table 5. We set the default batch size to 128KB (i.e., 8000 tuples), the default number of distinct keys to 1000, and the default key distribution to uniform (i.e., by setting the Zipf exponent to 0). We also set the default parallelization strategy to inter-window parallelization and the default progressiveness mode to the lazy mode. Except in the result materialization experiment (Sec 4.5), we do not write the join matches to the sink's output buffer and only count the number of join matches. In our later experiment, we state which parameters we change and report the impact on the performance of SJAs. Furthermore, we run our experiment using tuples generated by our data generator. Each tuple consists of 4 bytes key, 4 bytes payload value, and 8 bytes timestamp arranged in a row-oriented layout.

3.4 Hardware

We execute our experiment on a machine with 2x Intel Xeon Gold 5115 processor (2.4Ghz 20 cores, with Intel Hyper-Threading Technology) arranged in two sockets, 192 GB of main memory, and an Nvidia Tesla V100 with 16GB GPU memory. We also

[1] https://github.com/TU-Berlin-DIMA/gpu-stream-join-benchmark

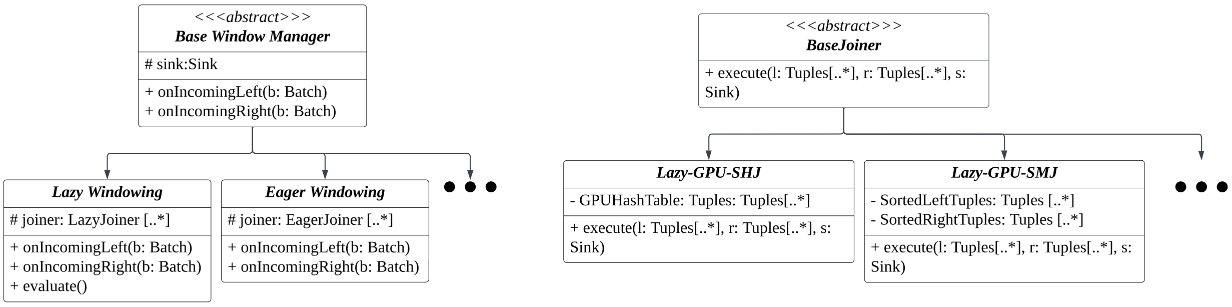

<<<abstract>>>
Base Window Manager
# sink:Sink
+ onIncomingLeft(b: Batch)
+ onIncomingRight(b: Batch)
Lazy Windowing
# joiner: LazyJoiner [..*]
+ onIncomingLeft(b: Batch)
+ onIncomingRight(b: Batch)
+ evaluate()
Eager Windowing
# joiner: EagerJoiner [..*]
+ onIncomingLeft(b: Batch)
+ onIncomingRight(b: Batch)
<<<abstract>>>
BaseJoiner
+ execute(l: Tuples[..*], r: Tuples[..*], s: Sink)
Lazy-GPU-SHJ
- GPUHashTable: Tuples: Tuples[..*]
+ execute(l: Tuples[..*], r: Tuples[..*], s: Sink)
Lazy-GPU-SMJ
- SortedLeftTuples: Tuples [..*]
- SortedRightTuples: Tuples [..*]
+ execute(l: Tuples[..*], r: Tuples[..*], s: Sink)

Figure 6: Windowing and SJA interface in our benchmarking framework.

execute several experiments on an Nvidia Jetson AGX Xavier with 512 CUDA cores and 4GB memory. We compile our benchmarking framework using g++ 9.3.0 and run the experiment on Ubuntu 22.04 LTS with CUDA Toolkit 11.7.

4 EVALUATION

In this section, we evaluate the impact of different parameters on the performance of SJAs. In particular, we evaluate the impact of data stream characteristics on different underlying join algorithms (Sec 4.1), the impact of parallelization strategies (Sec 4.2), the impact of algorithm progressiveness (Sec 4.3), the energy efficiency (Sec 4.4) and result materialization method (Sec 4.5). In addition, we also evaluate state-of-the-art GPU-accelerated SJAs using our benchmarking framework in Section 4.6.

4.1 Join Algorithm

The different underlying operations of SHJ and SSMJ make the two algorithms sensitive to different workload characteristics. Although there have been numerous works that compare the performance of HJ and SMJ for data at rest [2, 9, 17], the evaluation in previous work for streaming fashion [39] is limited to specific cases. Hence, in this section, we conduct a more in-depth investigation into the impact of varying data characteristics on the performance of SHJ and SSMJ. In particular, we evaluate the impact of numbers of distinct keys (Sec 4.1.1) and skewness (Sec 4.1.2) of the join keys.

4.1.1 Distinct Keys. In this experiment, we evaluate the sensitivity of SHJ and SSMJ to the number of distinct keys in the streams. To this end, we execute stream join queries on streams with varying numbers of distinct keys and measure the maximum sustainable throughput of CPU and GPU variants of SHJ and SSMJ. In particular, we use distinct key values of 0.1K, 1K, and 10K keys. Our preliminary experiment shows that the result of other distinct keys generalizes to these values. In addition, we run the experiment on different window sizes to vary the workload size and reveal its performance implications in relation to the number of distinct keys.

Figure 7 shows the maximum sustainable throughput achieved by the CPU and GPU variants of the SHJ and SSMJ on different numbers of distinct keys. On the x-axis, we show the window sizes, and on the y-axis, we show the maximum sustainable throughput of each algorithm. Our observations are as follows. First, a higher number of distinct keys lead to a higher maximum sustainable throughput in CPU-SHJ and GPU SHJ. This pattern is present across different window sizes. When joining streams with 10K distinct keys, GPU-SHJ outperforms GPU-SSMJ, i.e., achieving up to 9% higher throughput. In contrast, CPU-SSMJ and GPU-SSMJ tend to achieve similar performance for different numbers of distinct keys. Second, Figure 7 also shows that using GPUs does not always lead to a higher sustainable throughput. For instance, CPU-SSMJ achieves 24 − 33× higher throughput than GPU-SHJ when processing streams with 100 distinct keys. Thus, understanding the suitability of an algorithm to the characteristic of the workload is an essential step before offloading its processing to GPUs. To reveal the driving factors of these performance characteristics, we break down the execution time as follows.

Breakdown. Table 6 shows execution time breakdowns of GPU-SHJ and GPU-SSMJ in processing stream joins with different numbers of distinct keys on a window size of 4 MB. We observe that GPU-SHJ shows 2.9 − 5.8× longer average probe time in processing streams with an order of magnitude fewer

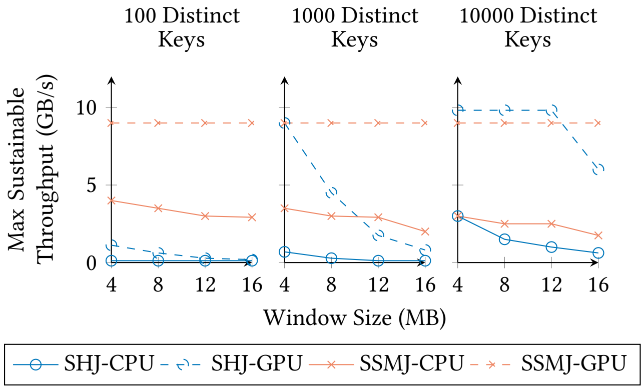


Figure 7: Performance of Hash Join and Sort-Merge Join on different numbers of distinct keys.

Table 6: Execution time breakdown of GPU-SHJ and GPU-SSMJ on different numbers of distinct keys.

Distinct Keys	GPU-SHJ		GPU-SSMJ	
	Build	Probe	Sort	Merge
0.1K	0.84 ms	5.57 ms	1.68 ms	0.05 ms
1K	0.71 ms	0.95 ms	1.51 ms	0.06 ms
10K	0.9 ms	0.33 ms	1.47 ms	0.08 ms

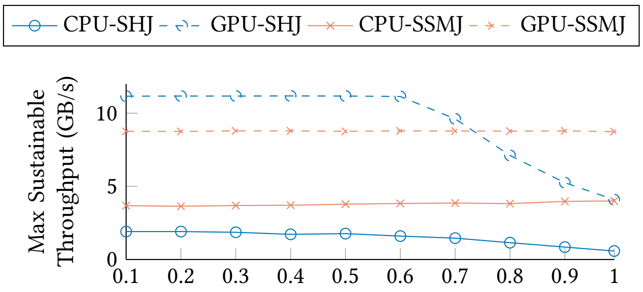


Figure 8: Performance of Hash Join and Sort-Merge Join for different degrees of skewness.

number of distinct keys. Lower numbers of distinct keys generate larger buckets, which take longer to probe. In contrast, different numbers of distinct keys only show a minor impact on the sort phase of GPU-SSMJ, which takes most of its computation time.

Summary. SSMJ is more robust against the number of distinct keys compared to SHJ, both in its CPU and GPU variants. In addition, our experiment also shows that CPU-SSMJ can reach higher throughput than GPU-SHJ, which indicates that using GPUs does not always result in better performance. This finding further emphasizes the importance of understanding the workload characteristics before offloading stream join workload to GPUs.

4.1.2 Skewness. In this experiment, we investigate the impact of different key skewness on the performance of SJAs. To this end, we vary the skewness of the input stream by choosing different Zipf exponents [10] in our data generator, i.e., higher Zipf exponents generate more skewed join keys in our input tuples. Following our preliminary experiment, we set the window size to 2 MB, i.e., a moderate window size, such that the impact of window size scaling does not interfere with the impact of skewness. As for the performance metric, we measure the maximum sustainable throughput achieved by the CPU and GPU variants of SHJ and SSMJ.

Figure 8 shows the performance comparison of CPU and GPU variants of SHJ and SSMJ on different degrees of skewness. The x-axis shows the different Zipf exponent and the y-axis shows the maximum sustainable throughput achieved by each algorithm. SSMJ is more robust to skewness in the join key than SHJ in both CPU and GPU variants. In the case of CPU variants, the throughput of CPU-SHJ decreases with higher degrees of skewness, i.e., from 1.91 GB/s to 0.58 GB/s when the Zipf exponent is increased from 0.1 to 1. In contrast, the throughput of CPU-SSMJ remains similar with higher degrees of skewness, i.e., between 3.68 GB/s at a Zipf exponent of 0.1 and 4.0 GB/s at a Zipf exponent of 1. In the case of GPU variants, GPU-SHJ and GPU-SSMJ show a similar pattern to their CPU variants. GPU-SHJ also achieves lower maximum sustainable throughput on higher degrees of skewness. However, the throughput drops at a higher rate, i.e., from 11.14 GB/s at a Zipf exponent of 0.6 to 4.09 GB/s at a Zipf exponent of 1. The throughput of GPU-SSMJ remains around 8.75 GB/s throughput different degrees of skewness. In the following, we break down the execution time of GPU-SHJ and GPU-SSMJ to further investigate how skewness in the join key impacts the throughput of the two algorithms.

Breakdown. Table 7 shows the average build and probe time of GPU-SHJ and the average sort and merge time of GPU-SSMJ.

Table 7: Execution time breakdown for different degrees of skewness.

Zipf Exponent	GPU-SHJ		GPU-SSMJ	
	Build	Probe	Sort	Merge
0.1	0.42 ms	0.29 ms	0.82 ms	0.05 ms
0.8	0.45 ms	0.62 ms	0.87 ms	0.05 ms
1	0.50 ms	1.15 ms	0.88 ms	0.05 ms

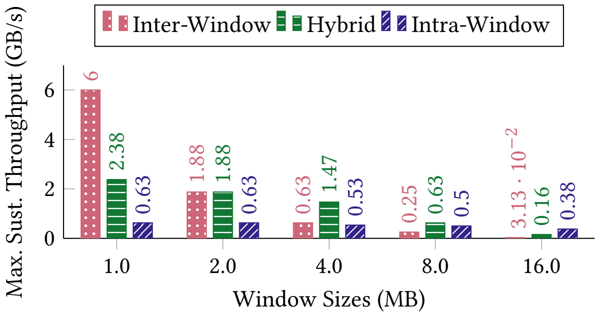


Figure 9: Maximum sustainable throughput of the Intra-Window, the Inter-window, and the Hybrid parallelization strategies on the CPU.

In GPU-SHJ, the longer average probe time accounts for the lower maximum sustainable throughput when the distribution of the join key is skewed. A skewed distribution of join keys leads to up to 4× higher average probe time on GPU-SHJ, i.e., from 0.29 ms to 1.15 ms with an increase of the Zipf exponent from 0.1 to 1. In this case, a higher number of GPU threads leads to larger buckets to scan on the probe phase. GPU-SSMJ only shows a marginal increase in average sort time with higher degrees of skewness. In particular, an increase of Zipf exponent from 0.1 to 1 only increases the average sort time from 0.82 ms to 0.88 ms. Thus, the maximum sustainable throughput of GPU-SSMJ drops at a lower rate compared to GPU-SHJ on increasing degrees of skewness.

Summary. Different degrees of skewness induce a similar impact on the performance of SJAs as the impact of different numbers of distinct keys. Thus, in this case, **SSMJ is also more robust to different degrees of skewness compared to SHJ**. However, SHJ algorithms can outperform SMJ algorithms in specific use cases, i.e., when the join keys are more uniformly distributed. Thus, we suggest practitioners to use SHJ for uniformly distributed keys and use SSMJ when the join keys are skewed.

4.2 Parallelization Strategy

Existing SJAs typically implement the Intra-window parallelization strategy, irrespective of the computational workload of the underlying stream join operation [15, 18, 19, 23, 39]. In this experiment, we compare the performance of the Intra-window parallelization strategy against two alternatives, i.e., the Inter-window and the Hybrid parallelization strategies. In particular, we analyze the throughput of these three parallelization strategies on an increasing window size from 1 MB to 16 MB and using the default values for other parameters.

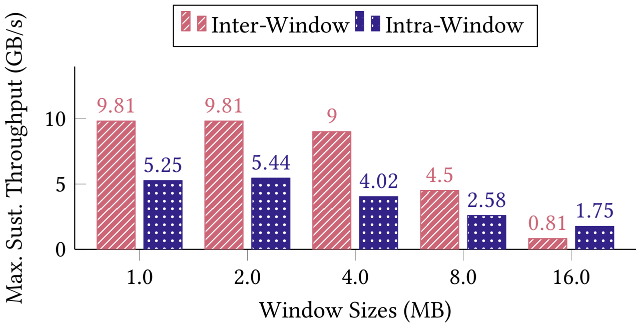


Figure 10: Maximum sustainable throughput of Intra- and Inter-window parallelization strategies on GPU.

4.2.1 *Impact on CPU.*

Figure 9 shows the performance of the three parallelization strategies on the CPU. Only using the Intra-window parallelization strategy prevents the SJA from achieving higher maximum sustainable throughput on relatively small windows (for example, for a window size of 1MB). In contrast, the Inter-window parallelization strategy allows the SJA to achieve a higher throughput on smaller window sizes (i.e., up to 6.0 GB/s), despite decreasing as the window size increases. The hybrid strategy offers a middle ground. The throughput of the hybrid strategy on small windows is lower compared to the Inter-window parallelization strategy, i.e., 3.0 GB/s compared to 6.0 GB/s. However, the throughput of the hybrid strategy decreases at a slower rate compared to the Inter-window parallelization strategy as the window size increases. Thus, at window sizes of 4MB and 8MB, it achieves the highest maximum sustainable throughput compared to the Inter-window and Intra-window parallelization strategies.

4.2.2 *Impact on GPU.*

State-of-the-art GPU-accelerated SJAs parallelize their computation on the GPU by employing the Intra-window parallelization strategy [15, 18, 19, 23, 39]. However, as an alternative, we can also use the Inter-window parallelization strategy to preemptively collect tuples belonging to a window and transfer the tuples to the GPU memory. Thus, the GPU can immediately evaluate the stream join on one window after another.

In Figure 10, we show the performance of the Intra-window and the Inter-window parallelization strategies on GPU. The Intra-window strategy only accelerates the SJA in one dimension, i.e., by shortening the window processing latency. As a result, the Intra-window parallelization strategy on GPU achieves 4.7 – 8.6× higher throughput compared to on CPU. However, the advantage diminishes as the window size gets smaller. At a window size of 1 MB, the throughput of the Intra-window parallelization strategy on GPU is even lower than on CPU with the Inter-window parallelization strategy, i.e., reaching only 5.25 GB/s compared to 6.0 GB/s.

In contrast, applying the Inter-window parallelization strategy allows optimization on another dimension. In the case of the Inter-window parallelization strategy, we allow the SJA to utilize all CPU threads to collect incoming tuples for a window and transfer it to the GPU while the GPU evaluates the kernel of other windows. As a result, the inter-window parallelization strategy achieves 1.2× to 1.86× higher throughput compared to the Intra-window parallelization strategy. This optimization is especially important in the case of small windows when the transfer and the computation take a similar time. In the case of the Intra-window parallelization strategy, this optimization is not possible as we only have a single thread receiving tuples and performing the windowing logic.

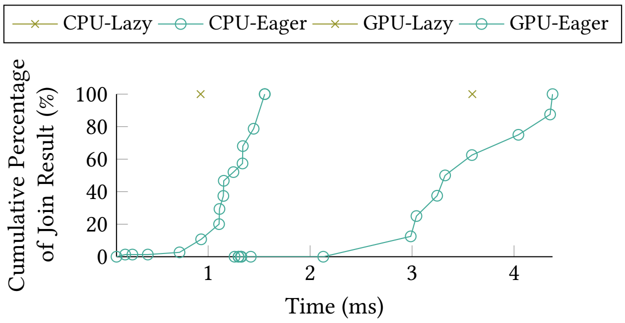


Figure 11: Result production time of the Lazy and the Eager progressiveness modes.

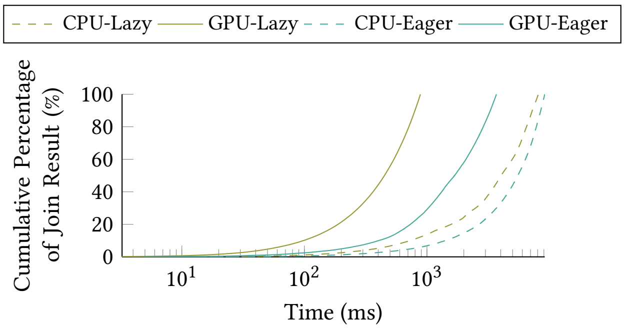


Figure 12: Cumulative percentage over an extended duration of stream.

Summary. Our experiment shows that only using the Intra-window parallelization strategy without considering the computational workload leads to suboptimal performance. **The Inter-window parallelization strategy allows us to optimize stream join on small windows.** When a GPU is available, we recommend offloading the Intra-window processing to the GPU as performed in state-of-the-art SJA in conjunction with the inter-window parallelization strategy to prevent the GPU from waiting for data to process.

4.3 Progressiveness Modes

In this section, we analyze to which extent the frequency of function invocation impacts the performance of SJAs. To this end, we run an experiment by executing a stream join workload using SHJ in the Eager and Lazy progressiveness modes. We implement a CPU and a GPU version for each mode, i.e., totaling four different variants: *CPU-Lazy*, *CPU-Eager*, *GPU-Lazy*, and *GPU-Eager.* All variants use intra-window parallelization, as the Eager mode only allows SJAs to operate on one active window at a time. We set other parameters to default and measure the cumulative percentage of produced join matches over time and the maximum sustainable throughput.

Figure 11 illustrates the impact of progressiveness mode on the cumulative percentage of produced join results over time for a single window. SJAs with the Eager progressiveness mode produce join results each time they receive a batch of tuples. In contrast, SJAs with the Lazy progressiveness mode wait for the

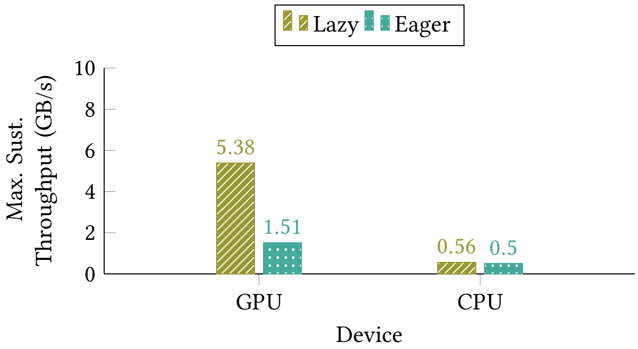


Figure 13: Throughput over an extended duration of stream.

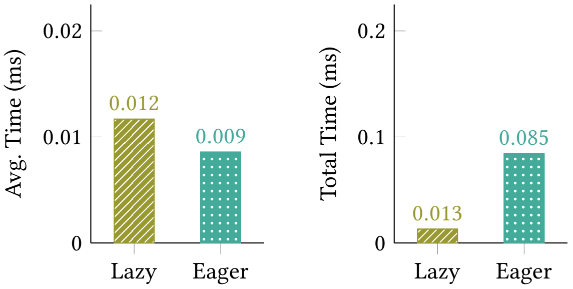


Figure 14: Kernel execution time breakdown of SJAs with the lazy and eager progressiveness mode.

entire tuples of the window to arrive and produce the join results once. Thus, we show it as a single point in the figure. Despite starting later than the Eager mode, the Lazy mode finishes the join computation of all tuples in the window earlier. Specifically, the CPU-Lazy takes 3.59 ms and CPU-Eager takes 4.38 ms, while GPU-Lazy takes 0.93 ms and GPU-Eager takes 1.56 ms.

In Figure 12 and 13, we scale our experiment on a longer duration of the stream. Our observations are twofold. First, the impact of progressiveness modes on the cumulative percentage of produced join results is greater, especially on the GPU, as we show in Figure 12. In this case, the number of join invocations in the Eager mode is even more frequent. Second, the more frequent kernel invocation on the Eager mode also leads to a lower maximum sustainable throughput on the GPU, as we show in Figure 13. GPU-Lazy achieves 3.56× higher throughput compared to GPU-Eager. In contrast, the throughput of CPU-Lazy and CPU-Eager are similar. To further study the reasons behind this pattern, we break down the execution time of GPU-Eager and GPU-Lazy as follows.

Figure 14 shows the total and average kernel execution time of GPU-Eager and GPU-Lazy. The join kernels of GPU-Lazy exhibit a longer average execution time than GPU-Eager due to performing build and probe operations at the window granularity. In contrast, GPU-Eager performs build and probe operations at the batch granularity, resulting in a lower average execution time. However, since GPU-Eager needs to perform more kernel invocations to produce all join results for the window, i.e., $2 * WindowSize/BatchSize$ (as it performs a symmetric hash join), its shorter average kernel execution time is not sufficient to compensate for the higher number of invocations it generates. Consequently, the total time of GPU-Eager is longer than that of GPU-Lazy.

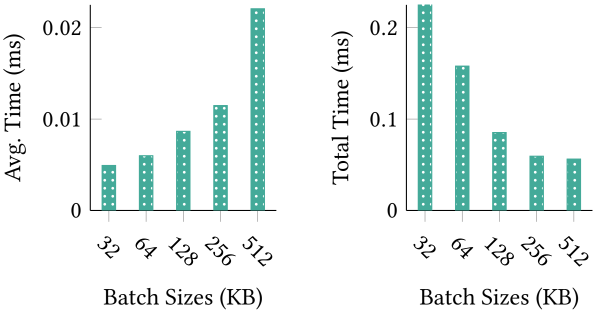


Figure 15: Impact of batch sizes to kernel execution.

In Figure 15, we extend this breakdown experiment on GPU-Eager with varying batch sizes, i.e., from 32 KB to 512 KB. Smaller batch sizes lead to shorter average kernel execution times, as it needs to process less number of tuples. However, smaller batch sizes require GPU-Eager to perform more kernel invocations. Similar to the comparison between GPU-Eager and GPU-Lazy, the shorter kernel execution time cannot compensate for the growth in the number of kernel invocations. Thus, it leads to a higher total kernel execution time on smaller batch sizes.

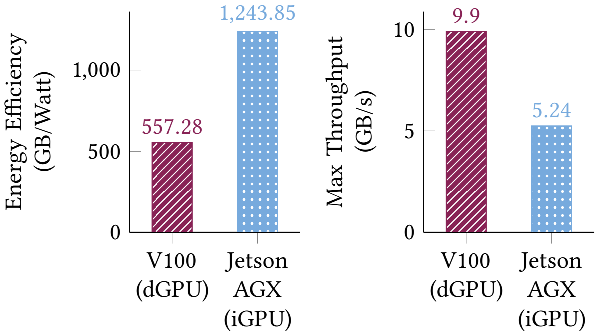
Overall, our findings suggest that a higher frequency of smaller workloads results in higher join execution latency. This pattern holds for progressiveness mode and batch sizes and other sources of small workloads, such as small window sizes. In such cases, smaller window sizes resemble smaller batches, triggering more invocations and leading to higher latency.

Summary. We show that **early results generated by the Eager progressiveness mode come at the price of a longer window processing time and a lower maximum sustainable throughput**. In addition, our results also detail the impact of progressiveness modes on SJAs' throughput and cumulative percentage of produced join results in the case of multiple windows. Thus, it further highlights the throughput difference between the Eager and the Lazy progressiveness, which was not discussed in previous work [39]. With this finding, we recommend using the Lazy progressiveness mode to achieve an overall higher cumulative percentage when executing stream joins and only opt for the Eager progressiveness mode on specific cases when knowing the first results of the join is necessary.

4.4 Energy Efficiency

The energy utilization of an algorithm is an important yet challenging issue. On the one hand, an energy-efficient algorithm could lead to lower operational costs, e.g., lowers electricity usage in a cloud infrastructure [16]. On the other hand, compute performance are directly proportional to energy utilization, i.e., high-energy utilization allows for faster processing [30]. This characteristic also applies to SJAs. Thus, it is essential to understand the energy efficiency of SJAs, especially when running on different hardware.

In this section, we investigate the energy efficiency of SJAs on two different classes of GPUs, i.e., dedicated GPU (dGPU) and integrated GPU (iGPU). To this end, we run an SHJ algorithm with a window size of 1 MB, using an inter-window parallelization strategy and lazy progressiveness mode. We measure energy efficiency as the amount of data that can be processed (in GB)



(a) Energy efficiency. **(b) Max sustainable throughput.**

Figure 16: Energy efficiency and maximum sustainable throughput on different GPU models.

per one Watt-hour(Wh). We measure the energy efficiency of an end-to-end stream join within the computing node, including the CPU, the GPU, and the interconnect between them. We also set the experiment parameter to allow the SJAs to reach their maximum sustainable throughput when measuring the energy efficiency of the SJAs.

Figure 16a shows the total processed data for every Watt-hour consumed by two GPU models. The Jetson-AGX integrated GPU produces more processed data per Watt-hour compared to the V100 dGPU, i.e., 1243.85 GB. In this case, the Jetson-AGX GPU operates on around 7 Watts to run the workload at its maximum sustainable throughput. The dedicated GPU, i.e., V100, shows significantly lower energy efficiency, i.e., 557.28 GB/Watt-hour. The V100 operates at around 65 Watts to run the workload at its maximum sustainable throughput. The power at which a GPU operates further impacts the runtime performance (e.g., maximum sustainable throughput) that can be achieved.

Figure 16b shows the throughput achieved by each GPU model on a stream join workload. Despite having lower energy efficiency, the higher power cap in the dedicated GPUs (i.e., 35 Watts and 65 Watts) compared to iGPU (i.e., 7 Watts) allows them to spend more Watt-hour in a given period. Consequently, dedicated GPUs achieve a higher maximum sustainable throughput. Specifically, the V100 achieves 10.0 GB/s. In contrast, the Jetson AGX can only achieve a maximum sustainable throughput of 1.75 GB/s.

Summary. In summary, iGPU has a higher energy efficiency, i.e., **iGPU can process a higher amount of data per Watt-hour at the price of lower maximum sustainable throughput**. Thus, iGPUs are suitable for cases when the required throughput is relatively low and when operating on energy-constrained devices. Conversely, dedicated GPUs deliver a higher maximum sustainable throughput and are suitable for cases with reliable energy sources, such as in the cloud environment.

4.5 Result Materialization

Existing GPU-accelerated SJAs employ various result materialization methods . However, these methods are tightly coupled and evaluated implicitly with the proposed SJA. Thus, their impact on the overall performance of the SJA remains unclear. This section examines how different result materialization methods affect the overall throughput of SJAs.

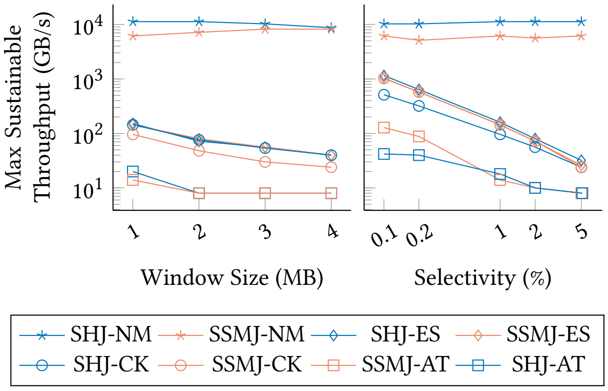


Figure 17: Impact of materialization result methods.

To this end, we implement SSMJ and SHJ using two result materialization methods, i.e., Count Kernel (*CK*) and Atomic (*AT*). In the case of *CK*, the algorithm creates a kernel to count the number of possible join matches, i.e., using the probe kernel in SHJ and the merge kernel in the SSMJ. Subsequently, it allocates an output buffer in the GPU to write the join result, write the join result to the buffer, and then copy the output buffer to the main memory. This method is similar to the result materialization performed in SABER [18]. In *AT*, the algorithm uses atomic operation to determine the offset in the output buffer to which it should write the join result. In this case, the output buffer is located in pinned main memory. Thus it does not explicitly perform a memory copy to the main memory. In addition, we implement two baselines in our experiment. First, we implement No Materialization (*NM*), which only counts the number of join matches and does not materialize them. Second, we implement Estimated Selectivity (*ES*), which assumes that the join selectivity is known and thus can be used to estimate the number of join matches. In this case, *ES* can then directly allocate the output buffer.

We set the SJA to use inter-window parallelization and lazy progressiveness mode. We run experiments on increasing window sizes from 1MB to 4MB and set other parameters to default.

In Figure 17, we show the impact of result materialization on the maximum sustainable throughput achieved by each SJA over an increasing window size. Result materialization accounts for up to three orders of magnitude difference in maximum sustainable throughput. In SHJ, the throughput of the NM variant ranges between 8.7 GB/s - 11.2 GB/s, while the ES variant, which adds only the materialization step, can only achieve a maximum sustainable throughput between 39.9 MB/s - 152 MB/s. Similarly, the ES variant of SSMJ can only achieve between 39.9 MB/s - 144 MB/s, while the NM variant achieves between 6.1 GB/s - 8.1 GB/s. The reason is that result materialization requires copying the join result to a result buffer (i.e., a buffer in the sink of the stream processing). The number of join results is significantly higher than the input, i.e., it can grow to $selectivity \times |Window| \times |Window|$. Hence, as the window size increases, the number of join results that need to be copied also increases and reduces the overall throughput of the SJA. Furthermore, in both SHJ and SSMJ, *AT* shows the lowest maximum sustainable throughput. *AT* copies the same amount of data to the CPU as *ES* and *CK*. However, *AT* copies the result at a tuple granularity and thus is inefficient.

Table 8: Parameters of State-of-the-art SJAs.

SJA	Join Alg.	Par. Strategy	Progres-siveness	Result Materialization
HELLSJoin	NLJ	Intra	Eager	Bitmap
SABER	NLJ	Intra	Lazy	Count Kernel
FineStream	NLJ	Intra	Lazy	Count Kernel
EcoJoin	HJ	Intra	Eager	No Materialization
EPHSA	HJ	Intra	Eager	Atomic

Figure 17 also shows that the CK the method leads to different performance patterns in SHJ and SSMJ. Specifically, CK further lowers the throughput of SHJ but does not impact the throughput of SSMJ. In SHJ, the probe kernel, which is used to compute the count of join matches, incurs random reads in the GPU memory and thus is inefficient as it prevents coalesced access to the GPU global memory. In contrast, the merge kernel of SSMJ works on sorted tuples and thus is more efficient and should be used when the number of join result of the join needs to be deduced as the stream progress. Thus, besides buffer copying, the inherent method from the underlying algorithm can further impact the extent of performance slowdown induced by result materialization. This finding enriches our understanding of the impact of result materialization, which is not discussed in prior work [18, 19]

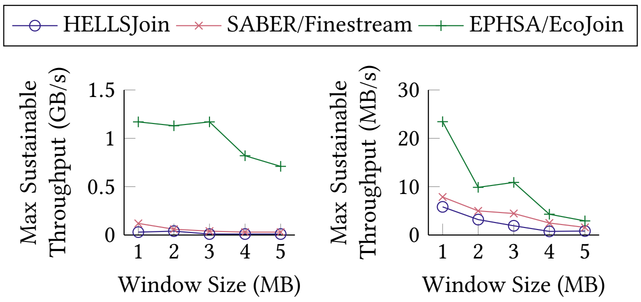
In addition to window size, the selectivity of the join also impacts the number of join matches. Figure 17 shows the maximum sustainable throughput of SHJ and SSMJ on different selectivities. A higher selectivity rate produces more join matches to materialize. Thus, we observe a similar pattern to Figure 17, i.e., lower the maximum sustainable throughput as the selectivity increases.

Summary. In summary, **join result materialization can be the bottleneck of SJA as it leads to up to one order of magnitude lower maximum sustainable throughput**. Thus, considering result materialization and changing data characteristics, practitioners should use SSMJ with a count kernel, as its merge kernel is more efficient compared to the probe kernel of SHJ.

4.6 Evaluating State-of-the-art SJAs

In this section, we model state-of-the-art SJAs using our benchmarking framework and evaluate the concrete instance of each model. To this end, parameterize the model by setting the join algorithm, the parallelization strategy, the progressiveness, and the result materialization method according to the proposed approach in the state of the art. In Table 8, we show the parameter of five state-of-the-art GPU-accelerated SJAs, including HELLSJoin [15], SABER [18], FineStream [38], EcoJoin [23], and EPHSA [19]. Each of these states of the arts represents a particular point in the design space of GPU-accelerated SJA. We measure the maximum sustainable throughput of each variant on an increasing window size from 1 MB to 5 MB and set other parameters to default.

In Figure 18a, we show the performance of state-of-the-art SJAs without result materialization. In this case, the different underlying algorithm leads to a major performance difference. EPHSA and EcoJoin use a hash join, and thus achieve a higher maximum sustainable throughput, compared to SABER, FineStream, and HELLSJoin, which are based on a nested loop join. In addition, the lazy progressiveness of SABER and Finestream leads to a slightly higher throughput compared to the eager progressiveness of HELLSJoin, i.e., following our finding in Section 4.3.



(a) Without materialization. (b) With materialization.

Figure 18: Maximum sustainable throughput of state-of-the-art GPU-accelerated SJAs.

Figure 18b shows the performance of state state-of-the-art SJAs with result materialization. Our observations are as follows. First, result materialization takes significant processing time on each SJA and results in orders of magnitude lower maximum sustainable throughput. At a window size of 1 MB, EPHSA sustains a maximum sustainable throughput of 1.17 GB/s without result materialization, and only 23.4 MB/s with result materialization. This result follows our finding in Section 4.5. Second, the choice of the underlying algorithm provides a higher impact compared to the result materialization method. Despite using the *Atomic* result materialization method, EPHSA, which is based on a hash join, achieves a higher maximum sustainable throughput compared to SABER and FineStream which are based on nested loop join and using the count kernel result materialization method.

Summary. In summary, **our benchmarking framework allows us to model state-of-the-art SJAs in the design space and evaluate them**. Our experiment results show that utilizing a more efficient underlying algorithm leads to higher performance differences compared to the choice of parallelization strategy and result materialization methods.

5 DISCUSSION

In this section, we summarize our findings from evaluating SJAs with different parameters. In particular, we provide a guideline to select a suitable variant of SJA for a given circumstance (Sec 5.1), describe our key lessons learned (Sec 5.2), and discuss open challenges in regarding GPU-accelerated SJAs (Sec 5.3).

5.1 Selecting a Stream Join Algorithm

In Figure 19, we provide a guideline to select a variant of SJA for a given setup. We decompose the processes of selecting a variant of SJA into four independent decisions: the progressiveness, the underlying join algorithm, the parallelization strategy, and the energy constraint. First, depending on the need for early result production, we can choose either the eager or the lazy progressiveness mode. The eager mode is more suitable than the lazy mode if the query requires the system to report early results. Otherwise, we can opt for the lazy progressiveness mode, which can sustain a higher ingestion rate. Second, we can choose either SSMJ or SHJ based on the characteristics of the event stream, i.e., the number of distinct keys and their distribution. We can choose the SHJ when processing streams with low numbers of distinct keys, which are uniformly distributed. In contrast, SSMJ is more

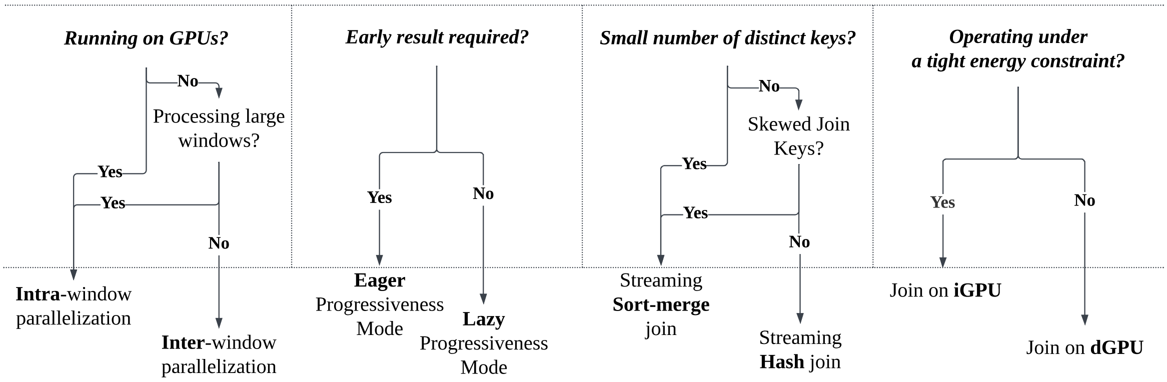

Running on GPUs?
No
Processing large windows?
Yes
Yes
No
Intra-window parallelization
Inter-window parallelization
Early result required?
Yes
No
Eager Progressiveness Mode
Lazy Progressiveness Mode
Small number of distinct keys?
No
Skewed Join Keys?
Yes
Yes
No
Streaming Sort-merge join
Streaming Hash join
Operating under a tight energy constraint?
Yes
No
Join on iGPU
Join on dGPU

Figure 19: A guideline on choosing a suitable stream join algorithm.

robust to different numbers of distinct keys and key distribution. Third, we can choose either intra-window or inter-window parallelization based on whether we target GPUs or CPUs and the size of the window we are working on. The intra-windows strategy is suitable when we only use CPUs to process large windows. Otherwise, choosing the inter-window strategy leads to a higher maximum sustainable throughput. Fourth, we can use dGPUs to sustain use cases with high ingestion rates and iGPUs for their energy efficiency. Furthermore, regarding result materialization, we recommend using the Count-Kernel method to achieve the maximum throughput. With the chosen option from each parameter, practitioners can then implement an SJA that best suits their query, data, and hardware characteristics.

5.2 Key Lessons Learned

In the following, we summarize further lessons learned from our experiments.

(1) GPUs can speed up SJA computation, but not in all cases. Our evaluation shows that GPUs are a feasible device to accelerate stream join processing. With the possibility to exploit both the intra-window and the inter-window parallelization strategies, GPU variants of SJA achieve up to 1.63× speed-up compared to the CPU variants. However, frequent kernel invocations, which are the consequence of using small batch sizes, processing small windows, or using an SJA with eager progressiveness mode, lead to lower throughput in GPUs than CPUs. Therefore, a careful examination of the characteristics of the workload is an essential step before offloading SJA processing to GPUs.

(2) SSMJ algorithms are more robust to data characteristics, but SHJ shows a higher peak performance. We observe that the SSMJ algorithm is more robust to skewed key distribution and a low number of distinct keys in the event stream compared to SHJ. However, in the case of GPU, our experiment shows that GPU-SHJ achieves up to 9% higher peak performance when processing streams with a large number of distinct keys (i.e., 10K keys).

(3) SJAs with the eager progressiveness mode produce early results but induce higher processing time. Performing the join processing tasks as soon as a batch of tuples arrives allows SJA with the eager progressiveness mode to produce join results early. Early production of join matches is useful for certain use cases, such as producing fast visualizations to decide actions for the rest of the join operation [7]. However, the eager progressiveness mode also incurs a high cost and thus results in poor performance, i.e., up to 1.2× longer execution time per window on CPUs and 1.7× on the GPU.

(4) Edge-grade GPUs achieve a lower throughput but are more energy efficient. The integrated GPU only achieves a fraction of the throughput of the dedicated GPU, i.e., by a factor of 5. However, iGPUs are more energy efficient both during active executions of stream joins and in idle time. Our experiment shows that iGPUs could process up to 2.2× data as dGPUs for the same Watt-hour consumed by the devices.

(5) Result materialization methods account for an order of magnitude lower maximum sustainable throughput. Despite writing the same amount of data, different result materialization method leads to different degrees of performance overhead. Our experiment shows that the Count-Kernel method achieves up to 10.2× a higher throughput than the Atomic method.

5.3 Open Challenges

On top of our findings, the direction of future SJAs on GPUs should focus on lifting assumptions taken by research prototypes [15, 18, 19, 23, 38]. Consequently, we should address the following resulting open challenges to allow swift integration of GPU-accelerated SJAs into SPEs and tackle real-world problems.

Stream I/O. We should lift the assumption that the event streams are already present in the system's main memory. SPEs typically accept event streams that are ingested through a network interface [4, 34, 36, 37]. The indirect communication between the network interface and the GPU may prevent us from supplying the GPU with data to process in a timely manner. Thus, we need to rethink how to directly ingest even streams to the GPU to fully reap the benefit of its acceleration.

System Concurrency. We should not assume that an SPE exclusively runs a single stream join query. Multiple streaming queries may run indefinitely and share the limited resources on a GPU. Therefore, we should consider the current system's load when parameterizing SJAs on GPUs. Furthermore, we also need to take care of efficient execution of concurrently running queries on a system.

Data Arrival Pattern. we should not assume constant characteristics of event streams throughout the lifetime of a query.

Real-world event streams may continuously change, have a sudden burst and drop in their ingestion rate, and arrive in an out-of-order manner. Thus, we need to adapt to the changing characteristics while the event streams continue to arrive without losing any data.

6 RELATED WORK

In this section, we describe related work and group them into three topics, i.e., GPU-accelerated stream processing engines, SJAs on GPUs, and existing performance comparisons of different GPU-accelerated SJAs.

GPU Accelerated Stream Processing Engines. The growing volume and velocity of data stream to process and the increasing availability of GPUs have led to the emergence of GPU-accelerated stream processing engines. In general, recent works in this area are either extending an existing system to utilize GPUs [6, 33] or creating a prototype for a new system [18, 19, 38]. The findings from both lines of work show a highly varied performance benefit from utilizing GPUs and thus are hard to interpret. In this paper, we focus our analysis on stream join queries and provide a practical guideline to help practitioners efficiently utilize GPUs for their stream join use cases.

Stream Joins on GPUs. There have been several approaches proposing the utilization of GPUs to accelerate the execution of stream joins. On the one hand, Karnagel et al. [15] propose HELLS-Join to execute band join on time-based sliding windows. Their approach leverages integrated GPUs and divide the workload between CPU and GPU. Similarly, Körber et al. [19], and Michalke et al. [23] also utilize the integrated GPU to accelerate stream join processing. On the other side, Kaliousis et al. [15] and Zhang et al. [38] use dedicated GPUs to accelerate stream joins. In both cases, the authors perform the comparison in isolation and only cover a narrow set of dimensions that impact the performance of GPU-accelerated stream joins. In this paper, we cover parameters from multiple dimensions (e.g., data, query, and hardware) and reveal their impact on the performance of stream join executions. We also extend the variety of underlying join algorithms in our analysis by including SMJ-based SJAs, parallelization strategies, and progressiveness modes. Furthermore, we provide a recommendation on selecting the best algorithm in a given circumstance. Thus, helping practitioners and system builders implement SJAs for a given environment.

Comparing GPU-Accelerated SJAs. The development of different techniques to perform stream joins on GPUs has led recent works to assess and compare their performance. Zhang et al. [38] compare the join performance of their SPE with Saber [18] using the same dataset and queries. In their evaluation, the authors show that CPUs deliver a higher throughput compared to integrated GPUs in executing stream join workloads. However, there is no discussion on the workload characteristics, and the result only gives us a snapshot of stream join performance for a single case. In our experiment, we show a more comprehensive view of the performance characteristics by considering both cases where either CPUs or GPUs achieve higher throughputs. In another research, Körber et al. [19] compare the stream join operator of their system with HELLSJoin [15]. In their experiment, the author shows that frequent kernel invocations may deteriorate the performance of GPU-accelerated SJAs. However, the authors only report the throughput and do not discuss whether the performance difference comes from the kernel invocation overhead. In our experiment, we isolate the problem of frequent kernel invocations and point out the case where it becomes the major performance-impacting factor. In general, apart from comparing limited numbers of algorithm variants, existing works use different workloads and devices and measure different metrics in their experiments. Thus, they are also not comparable. In our paper, we set up a common ground for different SJAs and measure the same metrics in different workload scenarios to reveal the characteristics of SJAs.

7 CONCLUSION

In this paper, we shed light on the configuration parameter space of SJAs and make existing approaches comparable. To this end, we investigated performance-impacting factors of GPU-accelerated SJAs and revealed their performance characteristics. In particular, GPUs are feasible for accelerating SJAs, but they require multiple parameters that must be carefully tuned for each workload and the underlying hardware. Based on that, we summarized the lessons learned and presented a simplified guideline to help practitioners to implement an SJA for their unique requirements. With this work, we lay the foundation for efficient GPU-accelerated SJAs.

For future work, we envision leveraging and integrating knowledge that we have learned from this paper into existing or next-generation SPEs (such as NebulaStream [36]). For instance, by incorporating the knowledge into a query compilation engine to generate code for stream join operators depending on the query, data, and hardware characteristics[11]. Furthermore, as our benchmarking framework is open-source, we enable other researchers to test their approaches in a representative setting against state-of-the-art solutions. Finally, we also envision extending the analysis to a wider range of hardware accelerators, such as DPU, HBM, and FPGA, which could bring benefit to more specialized use cases.

ACKNOWLEDGMENTS

This work was funded by the DFG Priority Program (MA4662-5), by the German Federal Ministry of Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A) and by Huawei as part of the project "Data Stream Processing on Heterogeneous Hardware". We thank the NebulaStream team for their insightful comments and fruitful discussions.

REFERENCES

[1] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. 2013. Photon: fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 577–588. https://doi.org/10.1145/2463676.2465272

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (2013), 85–96.

[3] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2014. GPU-Accelerated Database Systems: Survey and Open Challenges. *Trans. Large Scale Data Knowl. Centered Syst.* 15 (2014), 1–35. https://doi.org/10.1007/978-3-662-45761-0_1

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[5] Paris Carbone, Asterios Katsifodimos, and Seif Haridi. 2019. Stream Window Aggregation Semantics and Optimization. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Y. Zomaya (Eds.). Springer. https://doi.org/10.1007/978-3-319-63962-8_154-1

[6] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin A. Kwiat, and Charles A. Kamhoua. 2015. G-Storm: GPU-enabled high-throughput online data processing in Storm.

In *2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015*. IEEE Computer Society, 307–312. https://doi.org/10.1109/BigData.2015.7363769

[7] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer. 2002. Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 299–310. https://doi.org/10.1016/B978-155860869-6/50034-2

[8] Haralampos Gavriilidis, Adrian Michalke, Laura Mons, Steffen Zeuch, and Volker Markl. 2020. Scaling a Public Transport Monitoring System to Internet of Things Infrastructures.. In *EDBT*. 627–630.

[9] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. 1994. Sort versus Hash Revisited. *IEEE Trans. Knowl. Data Eng.* 6, 6 (1994), 934–944. https://doi.org/10.1109/69.334883

[10] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, Richard T. Snodgrass and Marianne Winslett (Eds.). ACM Press, 243–252. https://doi.org/10.1145/191839.191886

[11] Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2487–2503. https://doi.org/10.1145/3318464.3389739

[12] Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching, Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz, et al. 2018. Providing streaming joins as a service at facebook. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1809–1821.

[13] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. 2003. Evaluating Window Joins over Unbounded Streams. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman (Eds.). IEEE Computer Society, 341–352. https://doi.org/10.1109/ICDE.2003.1260804

[14] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1507–1518.

[15] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. 2013. The HELLS-join: a heterogeneous stream join for extremely large windows. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. 1–7.

[16] Avita Katal, Susheela Dahiya, and Tanupriya Choudhury. 2022. Energy efficiency in cloud computing data center: a survey on hardware technologies. *Clust. Comput.* 25, 1 (2022), 675–705. https://doi.org/10.1007/s10586-021-03431-z

[17] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[18] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. 2016. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*. 555–569.

[19] Michael Körber, Jakob Eckstein, Nikolaus Glombiewski, and Bernhard Seeger. 2019. Event stream processing on heterogeneous system architecture. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–10.

[20] Ramon Lawrence. 2005. Early Hash Join: A Configurable Algorithm for the Efficient and Early Production of Join Results. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 841–852. http://www.vldb.org/archives/website/2005/program/paper/thu/p841-lawrence.pdf

[21] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1633–1649. https://doi.org/10.1145/3318464.3389705

[22] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1017–1032. https://doi.org/10.1145/3514221.3517911

[23] Adrian Michalke, Philipp M. Grulich, Clemens Lutz, Steffen Zeuch, and Volker Markl. 2021. An Energy-Efficient Stream Join for the Internet of Things. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, Danica Porobic and Spyros Blanas (Eds.). ACM, 8:1–8:6. https://doi.org/10.1145/3465998.3466005

[24] Mohamed F. Mokbel, Ming Lu, and Walid G. Aref. 2004. Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, Z. Meral Özsoyoglu and Stanley B. Zdonik (Eds.). IEEE Computer Society, 251–262. https://doi.org/10.1109/ICDE.2004.1320002

[25] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 1, Article 11 (jan 2022), 38 pages. https://doi.org/10.1145/3485126

[26] Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2019. Performance Analysis and Automatic Tuning of Hash Aggregation on GPUs. In *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, Thomas Neumann and Ken Salem (Eds.). ACM, 8:1–8:11. https://doi.org/10.1145/3329785.3329922

[27] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel Index-based Stream Join on a Multicore CPU. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2523–2537. https://doi.org/10.1145/3318464.3380576

[28] Jens Teubner and René Müller. 2011. How soccer players would do stream joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 625–636. https://doi.org/10.1145/1989323.1989389

[29] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. 2014. Storm@twitter. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 147–156. https://doi.org/10.1145/2588555.2595641

[30] Dimitris Tsirogiannis et al. 2010. Analyzing the energy efficiency of a database server. In *SIGMOD*. ACM.

[31] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *VLDB J.* 32, 5 (2023), 985–1011. https://doi.org/10.1007/s00778-022-00778-6

[32] Annita N. Wilschut and Peter M. G. Apers. 1993. Dataflow Query Execution in a Parallel Main-memory Environment. *Distributed Parallel Databases* 1, 1 (1993), 103–128. https://doi.org/10.1007/BF01277522

[33] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. 2016. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (IEEE BigData 2016), Washington DC, USA, December 5-8, 2016*, James Joshi, George Karypis, Ling Liu, Xiaohua Hu, Ronay Ak, Yinglong Xia, Weijia Xu, Aki-Hiro Sato, Sudarsan Rachuri, Lyle H. Ungar, Philip S. Yu, Rama Govindaraju, and Toyotaro Suzumura (Eds.). IEEE Computer Society, 273–283. https://doi.org/10.1109/BigData.2016.7840613

[34] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[35] Steffen Zeuch, Sebastian Breß, Tilmann Rabl, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, and Volker Markl. 2019. Analyzing Efficient Stream Processing on Modern Hardware. *Proc. VLDB Endow.* 12, 5 (2019), 516–530. https://doi.org/10.14778/3303753.3303758

[36] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[37] Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M. Grulich, Ariane Ziehn, and Volker Markl. 2020. NebulaStream: Complex Analytics Beyond the Cloud. *Open J. Internet Things* 6, 1 (2020), 66–81. https://www.ronpub.com/ojiot/OJIOT_2020v6i1n07_Zeuch.html

[38] Feng Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2020. FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 633–647.

[39] Shuhao Zhang, Yancan Mao, Jiong He, Philipp M. Grulich, Steffen Zeuch, Bingsheng He, Richard T. B. Ma, and Volker Markl. 2021. Parallelizing Intra-Window Join on Multicores: An Experimental Study. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2089–2101. https://doi.org/10.1145/3448016.3452793