Hardware-Conscious Sliding Window Aggregation on GPUs

Georgios Michas
National and Kapodistrian University of Athens
Greece
g.michas@di.uoa.gr

Periklis Chrysogelos
EPFL
Switzerland
periklis.chrysogelos@epfl.ch

Ioannis Mytilinis
EPFL
Switzerland
ioannis.mytilinis@epfl.ch

Anastasia Ailamaki
EPFL, RAW Labs SA
Switzerland
anastasia.ailamaki@epfl.ch

ABSTRACT

Stream Processing Engines (SPEs) have recently begun utilizing heterogeneous coprocessors (e.g., GPUs) to meet the velocity requirements of modern real-time applications. The massive parallelism and high memory bandwidth of GPUs can significantly increase processing throughput in data-intensive streaming scenarios, such as windowed aggregations. However, previous research only focused on the overall architecture of hybrid CPU-GPU streaming systems and the need for efficient in-GPU window operators was overshadowed by the limited interconnect bandwidth.

With aggregation taking up a significant portion of streaming workloads, in this work, we analyze and optimize the performance of sliding window aggregates over GPUs. Current implementations under-utilize the hardware, and for a range of query parameters they cannot even saturate the bandwidth of the interconnect. To optimize execution, we first evaluate the fundamental building blocks of streaming aggregation for GPUs and identify the performance bottlenecks. Then, we build Slider: an adaptive algorithm that selects the most appropriate primitives and kernel configurations based on the query parameters. Our evaluation shows that Slider outperforms previous approaches by 3×-1250×, and saturates both the interconnect and the memory bandwidth for a wide range of examined input workloads.

KEYWORDS

stream processing, GPU, hardware, sliding window, aggregation

ACM Reference Format:
Georgios Michas, Periklis Chrysogelos, Ioannis Mytilinis, and Anastasia Ailamaki. 2021. Hardware-Conscious Sliding Window Aggregation on GPUs. In *International Workshop on Data Management on New Hardware (DAMON'21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3465998.3466014

1 INTRODUCTION

Real-time analytics usually involve the execution of continuous queries over high-throughput streams. To handle velocity and meet the performance requirements of such streams, common stream processing engines (e.g., Spark [9], Flink [1], Storm [8]) increase data parallelism by scaling-out to multiple machines. However, it has been shown that these systems underutilize the available hardware and cannot scale up efficiently [10]. Moreover, modern servers are usually heterogeneous and apart from multi-core CPUs, they are also equipped with hardware accelerators such as GPUs. To fill this gap, and exploit the parallelism provided by the hardware, there have been efforts to synergistically process data streams on both CPU and GPU cores [2, 4, 11]. These new, hybrid systems propose end-to-end solutions that integrate both architectures and efficiently schedule tasks to either of the two. Nevertheless, they do not delve into individual operators and do not discuss GPU-efficient algorithms for streaming operators, such as window aggregations, one of the biggest class of streaming workloads [7].

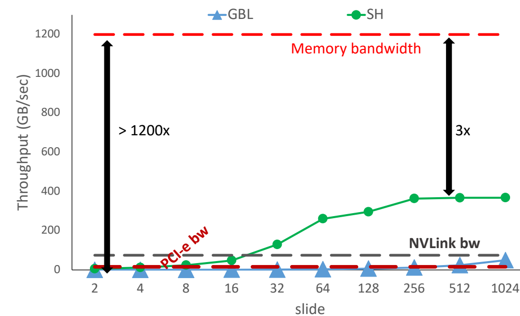

Figure 1: Hardware underutilization for a SUM query over a sliding window of 2048 items, and for various window slides. Execution takes place on an NVIDIA Tesla V100S GPU.

While efficient window aggregations for CPUs have been the focus of past work, GPU solutions are still underutilizing the hardware and for a range of configurations they incur more overheads than fetching data from the CPU. Figure 1 shows the missed opportunities for a simple `SUM` query over a sliding window of 2048 items, when we use two standard implementations (described in detail in Section 3). Performance varies depending on the window slide, but in all cases, the maximum achieved throughput is from 3× to more than 1200× lower than the memory bandwidth of the device. While window aggregations can theoretically achieve near memory bandwidth performance, existing approaches severely underutilize the resources, wasting GPU time from other more appropriate tasks, restricting the amount of window configurations that can be computed per data stream, and, for small *slide*s, they even fail to sustain the incoming data bandwidth, by running slower than the PCI-e or NVLink bandwidth.

This work analyzes the shortcomings of existing approaches, frames the fundamental building blocks of in-GPU window aggregation, and proposes Slider: an algorithm that automatically selects the best combination of building blocks based on the query and kernel configuration. Slider reduces the inter-thread communication,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAMON'21, June 20–25, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8556-5/21/06…$15.00
https://doi.org/10.1145/3465998.3466014

which is essential due to the massive GPU parallelism and limited per-thread cache. Furthermore, Slider, whenever possible, pulls operations to faster tiers of the memory hierarchy and cooperatively uses the GPU threads to avoid conflicts.

The contributions of this work are summarized as follows:

- We analyze the behavior of sliding window aggregation on GPUs and classify the challenges that arise due to query and hardware characteristics.
- We propose three building blocks that efficiently utilize the fast memory layers and reduce redundant computation through cooperative processing.
- We show how to automatically select the most appropriate configuration for the query at hand. Our evaluation shows that Slider, the proposed algorithm, saturates the interconnect and outperform previous approaches by 3× to 1250×.

2 STREAMING AND QUERY MODEL

We consider a stream $S = \langle t_1, t_2, t_3, ...\rangle$ as an infinite sequence of tuples. Similar to previous work on streams and heterogeneous co-processors, the stream is partitioned into disjoint batches of size B and execution happens in a per batch basis. Each batch is transferred to the device over the PCI-e bus and then, it is processed fully in parallel. As we focus on the implementation of the operators and not on the overall system, in this work, we assume that data has already been transferred and is GPU resident. Henceforth, with *input* we refer to the batch that has been transferred to the device's memory and not to the original stream.

Window aggregation forms a sequence of finite subsets over the input and computes an aggregate for each of these subsets. Here, we only consider count-based, sliding windows. Sliding windows define a fixed distance between the start of two consecutive windows. This distance is called a *slide*, and by allowing it to be smaller than the window size, adjacent windows overlap and a single tuple may contribute to the aggregate of multiple windows. More specifically, for each tuple, we have to update $\lceil \frac{W}{s} \rceil$ window results, where W is the window size and s the slide. For the remainder of the paper, we refer to the term $\lceil \frac{W}{s} \rceil$ as the *fan-out* of the aggregation.

The last thing to define for window aggregation is the aggregate function. Functions with different properties allow different kinds of optimization. Aggregate functions are classified [3] into: *distributive*, *algebraic*, and *holistic*. This work focuses on optimizing distributive aggregations; algebraic ones follow trivially and we leave holistic functions for future work.

3 WINDOW AGGREGATION ALGORITHMS

Similar to the CPU case, window aggregations take place in three phases: scanning, pre-aggregation and final aggregation. Here, we propose a suite of algorithms for the computation of distributive functions on the GPU; we explain how each algorithm implements each of the aggregation phases and discuss its bottlenecks.

3.1 BASELINE IMPLEMENTATIONS

We consider two common algorithms for implementing sliding window aggregation on the GPU: one that uses only the global memory (GBL) and one that also uses the scratchpad (SH). Both underutilize the hardware, nevertheless our proposed hardware-conscious algorithm builds on top of their building blocks.

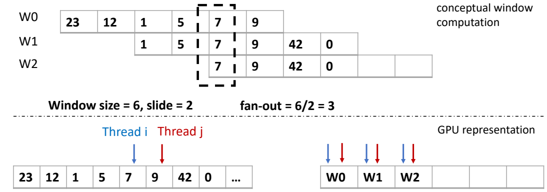

Figure 2: Window aggregation of size 6 and slide 2 and the corresponding layout of the GPU memory for GBL

GBL. The simplest way to compute a window aggregate is to scan the input from the global GPU memory, and for each item, to perform *fan-out* atomic updates to the output (also located in global memory). The process is illustrated in Figure 2. To exploit locality, avoid overfetching and maximize throughput, the scanning of the input happens in a coalesced manner. Naïvely, for each input item, the corresponding thread updates the affected output aggregates in a *sequential* manner. Thus, every s consecutive threads, read consecutive items from the input and update the same aggregate results in the output. Updates for the same aggregate are handled as conflicts and are serialized by the corresponding hardware units, leading to higher latency and lower throughput. To decrease such conflicts, we apply updates using a *circular* pattern: updates are arranged as a circular list and each thread consumes its list starting from the *lane id*-th update, reducing conflicts per update step.

SH. To remedy the high aggregation cost of GBL, we also consider an enhanced version of the baseline algorithm, that makes use of the local scratchpad. In GPUs, each streaming multiprocessor is equipped with a software-managed cache (scratchpad or in CUDA terminology "shared memory") that is shared among the threads of the same block and is much faster than global memory. Thus, we use this shared memory for pre-aggregation within a block. Each thread reads the input in a coalesced way and performs *fan-out* updates in the shared memory using atomic operations. Then, the partial aggregates of a block are flushed to the global output array.

By applying this pre-aggregation, the bottleneck shifts from updating the global memory to updating the aggregates in the scratchpad and flushing the reduced output. Shared memory is organized in banks and accesses to the same bank are serialized. To avoid overheads due to bank conflicts from intra-slide threads, we write using the circular pattern. Finally, as shared memory is a scarce resource, its capacity can be exceeded even for moderate $fanouts$, making this method inapplicable for high fan-outs.

3.2 GPU-SENSITIVE TRAITS OF WINDOW AGGREGATION

Window aggregations in GPUs can be decomposed into two logical execution phases: slicing and expansion. Similar to techniques such as PANE [5] for the CPU, slicing permits incremental computation. The expansion phase further combines the partial-aggregates in order to produce final results. This section analyzes the two phases, and matches their traits with the GPU hardware components.

3.2.1 Slicing. The first logical phase aims at slicing the stream in a convenient way that allows incremental processing, avoids redundant computation and facilitates the communication between different threads. The fact that, in sliding windows, every s items update the same output aggregate, creates both opportunities for optimization, but also conflicts across consecutive elements.

By always performing fan-out updates per input item, both GBL and SH avoid the need for slicing. However, as in PANE, we observe that by pre-aggregating consecutive input elements, the number of updates can be reduced by a factor of s.

Pre-aggregation requires either thread synchronization or the same thread to read and process consecutive input elements. In GPUs, assigning consecutive items to the same thread creates strided access patterns. For large slide values, each thread of a warp accesses multiple cache-lines, leading to overfetching. Thus, while such a strategy minimizes thread communication, it wastes input bandwidth and the overall throughput degrades. On the other hand, coalesced scans require cross-thread and, more importantly, inter-warp communication to combine partial-results.

To avoid both synchronization and materialization overheads, we use a parallel reduction in the warp-level. Specifically, we perform a *read-compress* process: first, we scan the input using vectorized loads (i.e., quadruples of 32-bit values). The loaded quadruple is directly pre-aggregated, and the corresponding thread maintains only the partial aggregate. As consecutive quadruples of the input are handled by different threads, whenever the slide allows it, we apply a second level of parallel, in-register aggregation through shuffling. Empty slots, created by this shuffling, are continuously refilled with new input data. This way, we ensure that all threads will be active and busy in the expansion phase that follows. The process is repeated until each thread has aggregated s items.

While assigning a thread per slide reduces the inter-warp communication and materialization, if the number of slides becomes lower than the number of threads, the device is under-utilized and throughput drops. To avoid this and keep all threads busy, we limit the number of items per thread based on the GPU's characteristics.

3.2.2 Expansion. This processing phase uses the pre-aggregated partial results and computes the final aggregates. As in CPUs, we need to make multiple updates per item in order to realize a final result, we call it the expansion phase. In this section, with the term input we refer to the output of the slicing phase. We propose two techniques for the expansion: (i) a *prefix*-based, whose communication is unaffected by the $fanout$ but requires an extra pass over the input, and (ii) a *cooperation*-based, which requires a single-pass but depends on the capacity of the local scratchpad. The selection of the most beneficial technique depends on the query parameters. To simplify the description, for the remainder of the section we use a `SUM` function. However, our algorithms can be trivially generalized to other distributive functions.

CoopExp, our cooperative approach, takes advantage of temporal locality and allows threads to cooperatively update multiple aggregates, while avoiding bank-conflicts and overfetching. As intra-warp communication happens at clock-speed, we use a two step process that first combines updates issued by threads of the same warp, and then issues a reduced number of atomic updates to the output. To further improve the performance, we use the scratchpad as an intermediary, pre-aggregation buffer.

Consecutive warps handle consecutive slides and thus they share aggregates. This property, along with the use of shared memory, allows for an extra optimization: only the first and last $fanout$ aggregates need to be atomically updated in global memory. The rest aggregates can be safely updated through regular store operations.

To reduce the total amount of operations in global memory, we configure the number of slides handled by a single block to the maximum number of slides whose working set can fit in shared memory. When the fan-out is large enough and exceeds the capacity of shared memory, CoopExp is not applicable and for the case of invertible functions, our second technique comes into play.

PrefixExp removes the dependency to the $fanout$. As a first step, we convert the input array to an array of prefix-sums. This can be done in near memory-speed by using Merrill and Garland's algorithm [6], implemented in NVIDIA's CUB library. Then we scan the array of prefix-sums and for each window edge, the corresponding thread looks back and accesses the start of the window. Then, we subtract the start from the end prefix-sum. In contrast to CoopExp, the prefix-sum requires only $O(1)$ state communication across thread blocks consuming consecutive input chunks [6].

Summary. CoopExp maintains the running-windows through atomic operations and executes them on shared memory to hide the cost of concurrent updates. In contrast, PrefixExp avoids shared state, by effectively increasing the impact of each input item to the full input sequence instead of the impacted window, relying on the invertibility of the aggregate function. Effectively, PrefixExp enforces the materialization and scanning of the prefix-sum to avoid the shared-memory capacity limitation.

3.3 END-TO-END ALGORITHMS

To provide efficient GPU window aggregation, we combine the presented building blocks based on the query parameters and propose the following algorithms: (i)**COOP-OPT**. Combining the slicing approach with CoopExp allows a single-pass algorithm, that requires a single scan of the input but is restricted by the amount of available scratchpad capacity. (ii)**DIRECT-PREFIX** skips the slicing phase and directly applies PrefixExp expansion. This algorithm needs 2 passes and also, the reading pattern of the second pass depends on the slide and may access items that reside in different cache lines. (iii)**REDUCED-PREFIX**. To reduce overfetching, we also use a 3-phase algorithm that combines COOP-OPT with PrefixExp expansion. In cases where the fan-out is too large and COOP-OPT cannot be applied for window W and slide s, due to shared memory limitations, instead we run COOP-OPT with a slide equal to the maximum common divisor of W and s, that is less than the scratchpad's size. Conceptually, this is equivalent to a slicing phase. Then, we can apply the 2-level process of the PrefixExp expansion. (iv)**SLIDER**. To get the best out of the available hardware, we devise Slider: a simple heuristic algorithm to select the most appropriate execution path for the slicing and expansion phases. Specifically, when the function is non-invertible, we always select COOP-OPT, with a fall-back to GBL, in case the scratchpad capacity is insufficient. When the function is invertible, Slider uses a cost model that lets it decide among the aforementioned options.

4 EVALUATION

In this Section, we show: (i) how the proposed algorithms improve throughput over the baseline implementations, and (ii) how they compare to each other. Since the proposed approaches are invariant to data characteristics (e.g., selectivity, skewness in time), in the interest of space, all the experiments use a $4GB$-sized array

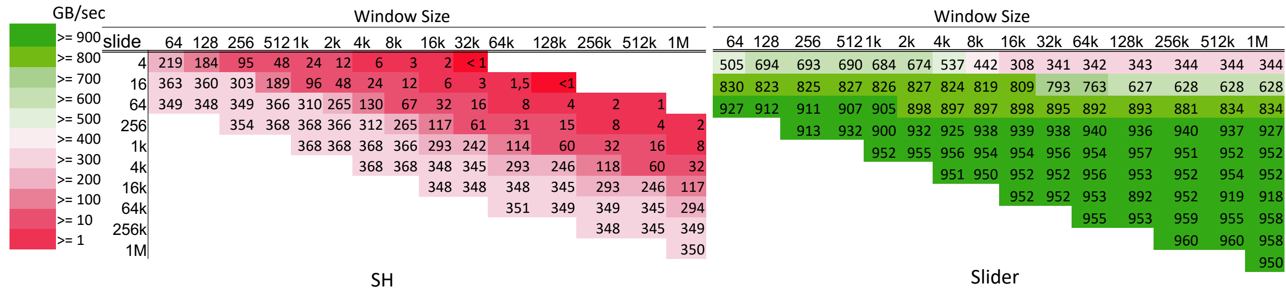

Figure 3: Throughput of SH and Slider for window sizes and slides in the range $[4, 1048576]$.

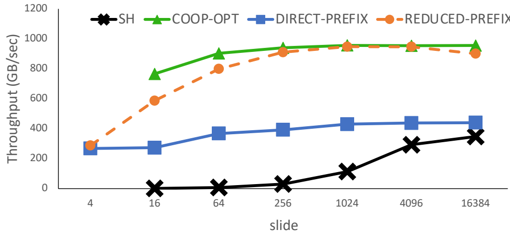

Figure 4: Comparison of COOP-OPT against the prefix-sum-based algorithms for a window of size $64k$ and various slides

of uniformly random, 32-bit integers and only distributive invertible functions. Specifically, we report performance only for `SUM` aggregations. However, the results for the baseline algorithms and COOP-OPT also hold for non-invertible functions, such as `MIN` and `MAX`. The determinant input characteristic is the window specification, i.e., the defined window size and slide. For this reason, both for the size and the slide, we exhaustively explore all powers of 2 up to 2^{20}, and we identify regions of the query space with different performance characteristics. For all the experiments we report throughput as consumed bytes (GB) over the execution time. Data are pre-loaded in GPU memory and as all the approaches have to read the full input to produce accurate results, their performance is upped bounded by the memory bandwidth.

All the experiments are conducted on an NVIDIA Tesla V100S GPU hosted in a 2 x 12-core Intel Xeon Gold 5118 CPU server with 376GB DRAM. The V100S GPU has 32GB of device memory, 1134GBps theoretical memory bandwidth and 84 SMs.

End-to-end algorithms. In Figure 4, we evaluate the proposed algorithms, and compare them with the SH baseline. DIRECT-PREFIX always outperforms the baseline, but as it requires two passes over the data and does not make any reduction, it cannot exceed the $440GBps$ (half the memory bandwidth) for any of the slides. In the case of REDUCED-PREFIX, the extra job seems to pay off: the larger the slide, the greater the potential reduction in the first job and the higher the achieved throughput, showing that the extra pass is significant only for small slides/reduction factors. Since COOP-OPT follows a multi-level pre-aggregation technique and requires a single pass over the data, it is always at least as fast as REDUCED-PREFIX. COOP-OPT outperforms REDUCED-PREFIX everywhere but in the small slides (high fan-out), where it cannot run due to shared memory limitations.

Figure 3(SH) shows the throughput of SH for different window configurations. Missing points in the bottom-left corner indicate that the slide cannot be greater than the window size. Colors change in a diagonal pattern, verifying that we hit bottlenecks based on the corresponding fan-out. Throughput never exceeds $368GBps$, and for the data-points in dark red color, the execution cannot even saturate the interconnect. Moreover, the missing points in the top-right corner indicate that the available shared memory capacity was insufficient and GBL should have been used instead. However, for the corresponding fan-outs GBL achieves less than $1GBps$, thus hardware underutilization is unavoidable for the baselines.

Slider algorithm. Figure 3(Slider) shows the throughput of the adaptive algorithm for each window configuration. For our GPU, and given the invertible nature of the used aggregate, Slider selects REDUCED-PREFIX whenever $fanout < 8K$, a little bit earlier than the $24K$ capacity limitation. Slider always exceeds the interconnect's bandwidth and almost always operates near the memory bandwidth, achieving more than $800GBps$ for 77% of the queries. Furthermore, comparing with the actual throughput of each approach, Slider never selected a suboptimal approach with a difference greater than noise levels.

5 CONCLUSIONS

In this paper, we improve the efficiency of sliding window aggregation on GPUs. In the past, the need for fast in-GPU streaming aggregation was overshadowed by the limited bandwidth of the interconnects. However, the technological advancements in interconnects push further the performance limits and stress the operators. In addition, there are query parameters for which sliding window aggregation runs below the PCI-e or NVLink bandwidth, failing to sustain the incoming data throughput. In order to fully utilize the available hardware, we decompose window aggregation into discrete phases and identify the fundamental characteristics, as well as the bottlenecks for each phase. Then, we map algorithmic characteristics to hardware components and create Slider: an algorithm that adaptively selects the kernel configuration and the right building block for each processing phase. Our evaluation shows that Slider outperforms existing approaches by 3× to 1250× and by operating at a throughput near the GPU memory bandwidth, for in-GPU data it is optimal and outperforms any CPU implementation.

6 ACKNOWLEDGMENTS

This work was partially funded by the EU H2020 project SmartDataLake (825041) and the SNSF project "Efficient Real-time Analytics on General-Purpose GPUs" subside no. 200021_178894/1.

REFERENCES

[1] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[2] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556. https://doi.org/10.14778/3303753.3303760

[3] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1, 1 (1997), 29–53.

[4] 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.

[5] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *Acm Sigmod Record* 34, 1 (2005), 39–44.

[6] Duane Merrill and Michael Garland. 2016. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[7] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment* 8, 7 (2015), 702–713.

[8] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 147–156.

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[10] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment* 12, 5 (2019), 516–530.

[11] 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 2020, July 15-17, 2020*. 633–647.