Toward GPU Accelerated Data Stream Processing

Marcus Pinnecke
Institute for Technical and Business Information Systems
University of Magdeburg, Germany
pinnecke@ovgu.de

David Broneske
Institute for Technical and Business Information Systems
University of Magdeburg, Germany
dbronesk@ovgu.de

Gunter Saake
Institute for Technical and Business Information Systems
University of Magdeburg, Germany
saake@ovgu.de

ABSTRACT

In recent years, the need for continuous processing and analysis of data streams has increased rapidly. To achieve high throughput-rates, stream-applications make use of operator-parallelization, batching-strategies and distribution. Another possibility is to utilize co-processors capabilities per operator. Further, the database community noticed, that a column-oriented architecture is essential for efficient co-processing, since the data transfer overhead is smaller compared to transferring whole tables.

However, current systems still rely on a row-wise architecture for stream processing, because it requires data structures for high velocity. In contrast, stream portions are *in rest* while being bound to a window. With this, we are able to alter the per-window event representation from row to column orientation, which will enable us to exploit GPU acceleration.

To provide general-purpose GPU capabilities for stream processing, the varying window sizes lead to challenges. Since very large windows cannot be passed directly to the GPU, we propose to split the variable-length windows into fixed-sized window portions. Further, each such portion has a column-oriented event representation. In this paper, we present a time and space efficient, data corruption free concept for this task. Finally, we identify open research challenges related to co-processing in the context of stream processing.

Keywords

Stream Processing; GPU; Large Windows; Circular Buffer

1. INTRODUCTION

Traditional Database Management Systems (DBMS) are designed to process a huge collection of data *in rest*. Queries are assumed to run once, deliver a single result set and then terminate immediately. This approach has been shown to be not suitable anymore to meet the requirements of new applications where *high velocity* data has to be processed *continuously* or *complex* operations are preformed on *high volume* data [1]. Real-time Stream Processing Systems (SPS) support *high velocity* and *high volume* [25] in combination with data flow graphs to achieve continuous computations over data streams in real-time. High throughput is achieved by distributed computing, parallelization, batching strategies and buffer management [1, 3, 21].

Research of the database community showed that the use of GPUs as co-processors are promising for data-intensive systems [14, 13]. Since a graphic card has a dedicated memory, each data to process has to be transferred from main memory to the graphic card memory and, after processing, vise versa. Since this transfer takes time, the transfer-cost might be a bottleneck for some applications [22]. A columnar DBMS is a suitable architecture for GPU acceleration, because it avoids transferring unneeded data (compared to a row store) and also favors data compression [5]. Since the content of a data stream changes rapidly, structured data-based SPSs process a stream of events as a stream of tuples [2, 3, 15, 21, 25]. Nevertheless, there is one fundamental buffering-technique called *windowing* that bounds a possible infinite data to finite portions. As stream-operators process a stream of windows either with a tuple-at-a-time or batch-at-a-time approach, we propose to focus on enabling efficient GPU accelerated operators for structured data in general and, hence, we address on-demand capabilities to convert regular length-variable windows into fixed-sized window portion-streams of columnar represented events.

In this paper, we address a strategy to enable this capability rather than a discussion of GPU-based operators itself. First, we show windowing in stream processing and graphic cards acceleration in DBMSs (Section 2). Afterwards, we continue with our main contributions:

- We examine a concept that splits any stream of variable-length windows into a stream of fixed-size window portions with columnar representation (Section 3)
- We identify open research challenges in context of co-processor-ready stream processing (Section 4)

We will finish with related work (Section 5) and our conclusion that sums up the paper's content (Section 6).

2. BACKGROUND

Motivated by the work of Karnagel et al., who showed a throughput increase for band join computations over streams using GPU acceleration [17], we believe that a general-propose GPU-ready stream processing framework should be established. The reasons are *(1)* it enables a single system for regular row-oriented stream processing via CPU and

27th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 26.05.2015 - 29.05.2015, Magdeburg, Germany.

Copyright is held by the author/owner(s).

efficient column-oriented stream processing via GPU and, hence, the load can be shared between both devices, *(2)* it allows us to run streaming applications on CPU, on GPU, or in mixed-mode and *(3)* it provides a further abstraction to achieve *what should be done* rather than *how should it be done*. Since time-based windows vary in length, it is not capable to send their contents directly to the GPU. Moreover, the current event presentation leads to memory usage overhead. To highlight these issues in more detail, we examine in the following stream processing and windowing as well as GPU acceleration in the context of DBMSs.

2.1 Data-Intensive Systems and GPUs

With the advent of general-purpose computing on graphics processing units (GPGPU), the usage of a GPU to solve arbitrary problems have become popular. Because of its high parallelism, a GPU can outperform a CPU by orders of magnitude. However, such performance improvements are only possible, if considering the special architecture of the GPU and its programming model.

2.1.1 GPU Architecture

Considering the GPU as an arbitrary co-processor for data processing, we have to care about two things: the *GPU execution model* and the *data transfer cost*. The overall execution model using OpenCL or CUDA uses a host process (on the CPU) and a *kernel* process (mainly a function with implicit parallelism) on the (co-)processor (here the GPU). The host with its host code manages data transfer and schedules the execution on the co-processor. To execute a kernel process on the GPU, there are four things to be done: *(1)* allocate enough memory for the input and output data, *(2)* copy the input data to the allocated device memory, *(3)* execute one or more kernel programs and *(4)* copy output data back to the host. Hence, batch-processing using a columnar event representation is the right choice for stream processing on the GPU, because the overhead of steps *(2)* and *(4)* will be too high otherwise.

2.1.2 Application Scope

Since data copying is an essential step to use a GPU, it is also the biggest bottleneck. For this, the host process (executed on the CPU) schedules the data transfer and provide necessary data over the PCI-Express Bus. However, compared to the bandwidth between the GPU memory and its cores, the bandwidth of the PCI-Express Bus is very low [5]. Hence, the GPU is most efficient for compute-bound problems, where data transfer plays only a minor role [22].

2.1.3 GPU Acceleration in Databases

Unregarded the high transfer costs, executing database operations on the GPU has shown notable performance benefits. For instance, the specialized sort implementation of Govindaraju et al. [13], GPUTeraSort, achieves a speedup of 10 compared to a CPU implementation. Also, specialized GPU join algorithms perform 3-8x better than reported CPU join implementations [16]. However, there are also operations, e.g., selections, that executed on the GPU may also harm performance, because they are not fully parallelizable [14].

Since not all operators benefit from GPU acceleration, further database research in the direction of load-balancing between co-processors is needed to get a benefit for the operators of a whole query plan, e.g., in CoGaDB [4]. This attempt relates to our effort in creating an adaptive system that also distributes its work between the CPU and GPU.

2.2 Stream Processing

Stream processing is a paradigm to continuously process, analyze and monitor a (possibly infinite) sequence of data, that is called a stream. Since traditional DBMSs assume *data in-rest*, exact results, queries initiated by humans and no real-time services for applications, they are not adequate for this task [1]. Therefore, around 2003, the database community researched adequate systems for stream processing. The academic projects *Aurora* and it's fork *Borealis* are notable here, because they provide an innovative model to deal with data streams and provide well-engineered strategies for scheduling, load shedding, high availability, high performance, and dynamic query management capabilities [1].

Since stream processing is data-driven, a user defined query consumes streams and produces streams containing the results. Those queries are *online* until they are terminated manually. This is in contrast to traditional DBMS queries, which terminate after their execution automatically. Those queries over streams are a loosely coupled data-flow network of operators.

2.2.1 Windowing

Since a stream is a possible infinite sequence of data, it is infeasible to store the complete input data in main memory. Therefore, one fundamental concept is a buffering technique called *windowing* that provides finite stream portions.

2.2.2 Window Types

There are many window variations due to the fact that there are many approaches about what to buffer (time-based or count-based for instance), how to handle new data, when to release old data, and when to trigger an operator [2, 3, 15].

If a window buffers a data stream on a *count*-based approach, the stream of outgoing windows has a fixed-length each. For instance, a count-based jumping window contains a fixed number of tuples and is updated after receiving a predefined number of new tuples. Hence, it contains tuples that occur in a variable time span such that the count of tuples in a window is stable.

On the other hand, if a data stream is buffered by a *time*-based approach, the tuple count inside a window varies. For instance, a time-based jumping window contains input data related to a given time span (e.g., five minutes) and is updated automatically after a certain time has passed, e.g. two minutes. Hence, it contains tuples that occur in a fixed time span such that the count of tuples in a window is variable.

Since a time-based approach is probably more common than its count-based counterpart [24], the varying length is critical in view of GPU-acceleration since *(1)* GPU memory is fixed-size allocated and *(2)* time-based windows might contain thousands of tuples when the time span is large enough and many events occur per instant. As a result, the complete batch might be too large for graphic card memory.

2.2.3 Large-Scale Window Management

We examine in the following large-scale window maintenance and processing as well as load distribution in traditional stream processing.

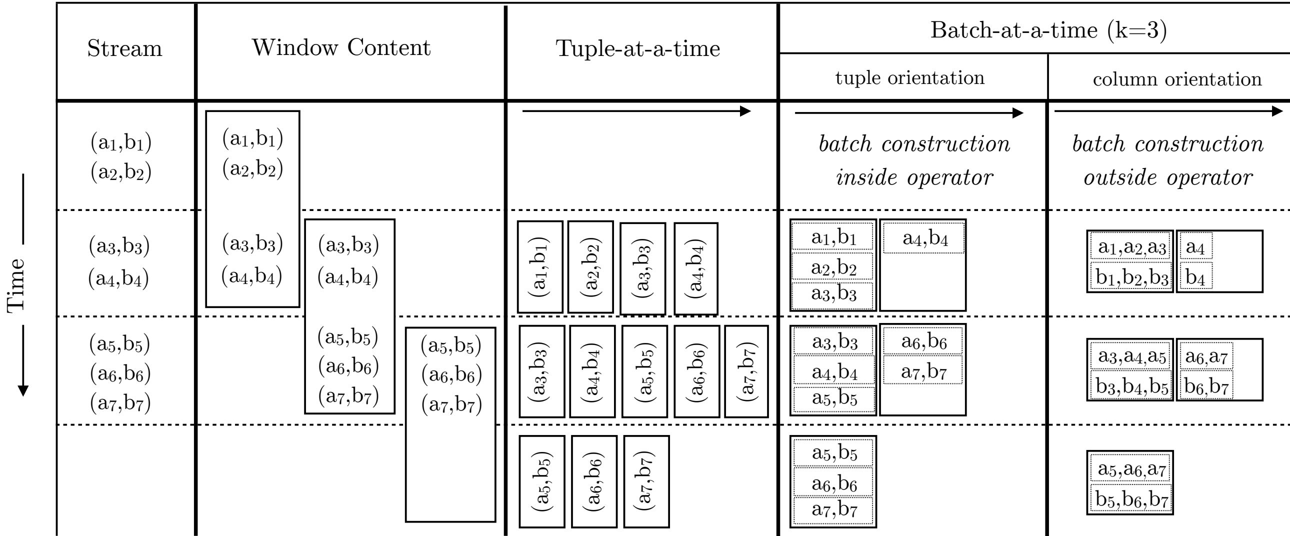


Figure 1: A continious stream of 7 events bound to a time-based window of size 2 instants. The Figure shows the processing using a tuple-at-a-time and batch-at-a-time approach. The latter shows row-orientation designed for CPU processing and column-orientation designed for co-processor usage.

Maintaining. To maintain large-scale windows in regular stream processing, the window might be partly swapped to disk. For instance, the storage manager in Aurora tries to hold the most relevant content in main memory while low prioritized ranges are paged to disk using a special replacement policy [1]. Besides *holding* large-scale windows, another important issue is to *process* these windows.

Processing model. One common approach is a tuple-at-a-time approach [15, 20, 21, 25] that does not care about the actual number of tuples in a window, since the required space depends on a single tuple. On the other hand, providing the ability to consume more than one tuple at a time using a batch-like approach [2, 11] could be achieved by iteration over the window's content inside the operator, utilizing the built-in RAM/disk swapping facilities. Here, an operator consumes k elements as a single block in each iteration, by setting some pointers into the window, until the window has been fully consumed [1]. Obviously, the event representation is unchanged, since it is optimized for CPU processing.

Figure 1 shows different processing approaches. The left hand side shows the content of a stream that is a sequence of $i = 1, ..., 7$ tuples (a_i, b_i) per time. Here, more than one tuple occur in one instant. The next column shows the actual window content, when applying a time-based window over two time instants. These windows are processed either in a tuple-at-a-time or batch-at-a-time processing manner. The left column for batching shows the actual construction that hold k tuples (row-orientated) per batch by the traditional stream processing approach.

Load Distribution. To increase the throughput per operator, load distribution strategies can be used such that the load to process a window can be shared. This could be achieved per-window (blocks are processed by parallel running operators [23, 25]) or per-block (a single operator processes partitions of each block in parallel [17, 19]).

3. GPU-READY STREAM PROCESSING

We introduce an operation that we call *bucketing* which transforms the output of windows such that they can efficiently consumed by GPU-based operators. To distinguish between what a stream-operator consumes, namely a window, we call what a GPU-operator consumes a bucket. We explore the differences in the following sub section. In contrast to windowing logic, these buckets are fixed-size in length independent from the window they listen to, such that we can pipe the window content bucket by bucket to the GPU-based operators. Moreover, we will examine how to flip the event representation from row to column and vise versa efficiently, to avoid unnecessary transfer costs to the graphic card.

We target fixed-sized bucketing of windows with a dedicated operator since this task should not be in the responsibility of any GPU-based operator for several reasons, such as redundant logic might occur otherwise. To show our strategy, we will introduce it step by step. We explore the differences to windowing first and show afterwards how buckets can be created efficiently. The latter approach is more general since we not address the representation-flipping here. How the transformation between row- and column-orientation could be achieved is explained afterwards.

3.1 Motivation

Since SPSs are tuple-oriented, their primary data structure to express an entity's content is a tuple. Although it is a common structure, it might not be efficient in terms of GPU data transfer cost and memory consumption. Consider for example a stream R with schema $\mathcal{R} = \{A, B\}$ where $\mathbf{dom}(A) = \texttt{char(50)}$ and $\mathbf{dom}(B) = \texttt{int}$. R could contain a tuple $(\text{the}, 268)$ for instance. Further assume a size of 54 bytes per tuple, where 4 bytes are used for the `int` data type. If an operation only requires B-values, sending entire tuples will waste $\approx 93\%$ of data transfer time and graphic card memory. Therefore, we focus on flipping the event

representation and bounding the size of windows to avoid out-of-memory situations when employing the graphic card.

3.2 Buckets vs. Windows

Windowing is one of the fundamental concepts of stream processing. To be clear about where our proposed operator is different, we examine similiarities and differences of buckets and windows.

First of all, the purpose of both concepts is different. Window operators are used to bound a (infinite) stream of data to a collection of data, called a window, that can be processed by set-oriented operators. Hence, a window operator consumes a stream of events and produces a stream of windows. As mentioned earlier, the actual window length can vary depending on a given policy, such as in the time-based approach. In contrast, our operator consumes a stream of windows, each might vary in length, and partitions each window into portions of a user-defined size k. At the same time, the row-wise representation is changed to columnar representation. Each such portion is called a bucket.

Assume a stream R with schema $\mathcal{R} = \{A, B\}$ and consider the right hand side of Figure 1. As one can see, regular batching with a batch-size $k = 3$ contains at most tree tuples $(a, b, c) \in R$ per batch entry, while a bucket of size three contains exactly $|\mathcal{R}|$ tuples each with most k components. These components per tuple are from the same domain, while the components of a window tuple are mixed-domains. Therefore, a bucket's tuple t with components from a domain T can forwarded directly to GPU since t is, sloppy saying, an array of length k and type T.

3.3 Portioning Variable-Length Windows

Whereas windowing a continuous data stream leads to several strategies to match different policies, bucketing a window is relatively straightforward. At any time, during runtime or initialization, the subscribers $S_1, ..., S_m$ that request to consume buckets are known. Also their individual bucket sizes $k_i = \mathbf{size}(S_i) \in \mathbb{N}^+$ are known. We receive a window ω of some size. Therefore, our task is to forward ω in portions of size k_i to each S_i until ω is completely consumed. We assume here w.l.o.g. that the bounds of each window are marked and that the length of each row-oriented tuple is fixed.

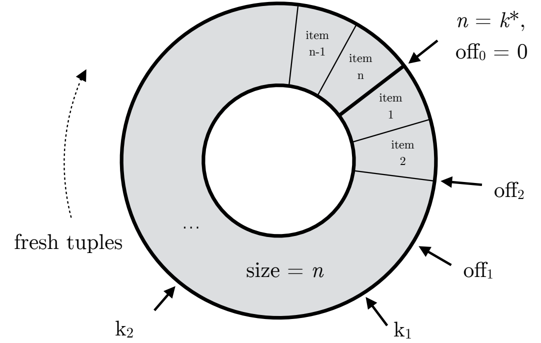
3.3.1 Algorithm

Let $k^* := \max\{k_1, ..., k_m\}$. We propose to use a fixed-size circular buffer CB that is able to hold items (here, tuple) at k^* positions. For each subscriber S_i we construct a range that we call $\mathbf{slice}_i := (\text{off}_i, k_i)$ that contains S_i's current offset off_i and its desired portion size $k_i > 0$. Each $\mathbf{slice}_i$ points inside CB – its range starts at off_i and ends at $\text{off}_i + k_i - 1$.

At initialization, we set $\text{off}_i = 0$ for all i and $h = 0$. Every time ω outputs a tuple, this tuple is inserted into CB. This insertion operation moves the internal head pointer h of CB further. Hence, h points to the position that will be written next. If $h = k^*$ holds, h is set to 0 since CB is a circular buffer. If after insertion the condition $(h \equiv \text{off}_i + k_i) \bmod k^*$ for a given subscriber S_i holds, the subscriber is notified. In case of a notification to S_i, S_i's offset off_i is moved to h.[1] When ω is completely read into CB, all S_i are finally notified

[1]Notably, each $\mathbf{slice}_i$ moves as a count-based jumping window of sized length k_i and jump size k_i over the content of ω until ω is fully consumed.

Figure 2: A circular buffer of fixed size k^* and three subscribers. Each subscriber $i \in \{0, 1, 2\}$ has it's own slice $\langle \text{off}_i, k_i \rangle$. Fresh tuples are inserted and slices are moved to partition a possibly infinite data stream. Hence, the stream is split into k_i-length portions that are forwarded to subscribers S_i



about the eventually remaining portion in $\mathbf{slice}_i$ and about the end of the operation. Afterwards each off_i is set to h such that it is reset to consume the next window. Figure 2 shows such a buffer for $m = 3$ subscribers.

If a new subscriber S_{i+1} occurs during runtime, we set its $\text{off}_{i+1} = h$. Since new data is more important than old data, S_{i+1} starts consequentially with an empty portion. However, if the bucket size of S_{i+1} is greater than the definied k^*, CB has to be resized.

3.3.2 Data Corruptions

Data corruption is overwriting data that has not been sent to subscribers. This is equivalent to a rearrangement of the time order of the data inside at least one $\mathbf{slice}_i$. This order is inducted by the order of the insertion operations into CB.

We claim that there can be no data corruption and proceed by contraposition. Assume using our construction and a data corruption. Then at least one $\mathbf{slice}_i$ contains two items a and b with b was inserted after a into CB but $\mathbf{slice}_i$ states a before b. This happens only if there is a confusion in $\mathbf{slice}_i$'s view, since data is written sequentially and ordered into a circular buffer CB and $\mathbf{slice}_i$ points inside CB. A confusion occurs, if CB's head pointer h moves one step after insertion, but from perspective of $\mathbf{slice}_i$, h moved more than one step. Consequentially, the range of $\mathbf{slice}_i$ is smaller than expected, since there should be an one-to-one mapping between the actual move of h and the perceived moved of h by $\mathbf{slice}_i$. This happens if the range of $\mathbf{slice}_i$ could not be embedded into CB at once and was wrapped around. That happens if CB is too small. This is a contradiction, since CB is as large as the longest requested portion in our construction. Therefore, there is no data corruption possible.

3.4 Flipping the Event Representation

We showed how to achieve an efficient window splitting without changes in the event representation in the previous sub section. Now, we will utilize our approach to flip row-oriented windows into a stream of column-oriented buckets.

Since the complexity of portioning any variable-length window is mainly driven by the actual window length, we suppose to utilize this concept for representation flipping.

The schema $\mathcal{S} = \{A_1, \ldots, A_n\}$ for the contents in a window is known. Hence, maintaining one circular buffer as in Section 3.3 for each attribute $A_i \in \mathcal{S}$ splits each incoming tuple into its components. We propose to use a *bucketing*-operator that is responsible for portioning the incoming windows such that it holds n circular buffers, each buffering one attribute of the incoming tuple. Since external subscribers $S_1, ..., S_\ell$ are known to the *bucketing*-operator, the operator adds itself as "internal" subscriber to each $1 \leq j \leq n$ circular buffers, delegates the desired $\mathbf{size}_i$ for each external S_i to all buffers. This leads to notification of all circular buffers to the operator at the same time once $\mathbf{size}_i$ is achieved for some external subscriber S_i. Now, the operator packages the n portions delivered to it into a collection, that is send to the external subscriber.

This allows a chain of operators where each consumes and produces a column-oriented event representation. To convert back to row-oriented event representation, another operator performs the described operations backwards. Hence, to revert the representation flip, another operator reads buckets and outputs them as a stream of regular tuples.

We propose to run these constructions in dedicated operators, since this allows sharing the costs of window portioning between several external subscribers and reverting the operation.

4. OPEN RESEARCH CHALLENGES

Based on existing research and related work, we cannot completely answer all relevant aspects. Therefore, we present two open research challenges in the following section.

4.1 Stream Processing on Modern Hardware

We propose an additional *bucketing*-operator to support event representation changes and window portioning to target GPU-specialized counterparts of existing stream operators. On the other hand, current trends in hardware bring further co-processors (e.g., Intel Xeon Phi or FPGA) with special characteristics into view. Consequently, these co-processors could be used to accelerate stream processing in addition to the GPU. With the increasing amount of different devices, we have to take care of optimized algorithms and execution models for the respected processor to also reach optimized performance [7, 8]. An essential part is to tune a given operator to the used (co-)processor, because each processing device has its own set of optimizations that can be applied [9]. Furthermore, with the availability of different devices, we have to decide where to execute a given operator to reach optimal performance. Here, it is not only important to find the device with the best performance for the given operator, but also to distribute the load between the devices similar to the work of Breß et al. [6]. As far as we can see, further research should be investigated to find limitations and benefits for applied modern hardware in this context.

4.2 Scheduler for Heterogeneous Devices

As proposed by Breß et al. [6], in the context of GPU acceleration for columnar databases, heterogeneous devices with dedicated memory are an opportunity since data transfer from one memory to another is optional. Utilizing OpenCL's possibility to execute kernels on different devices, Karnagel et al. suggest to use a unified kernel and a load balancer that partitions the incoming data and distributes them either to the CPU or GPU depending on the load [18]. This load balancer contains a job queue and a dictionary that maps tasks to several states. OpenCL takes free jobs as soon as a device is available and deploys it to a specific device on its own. An interesting question is, how an advanced load balancer improves execution performance even further, if the targeted device runs a specialized kernel. This could be achieved with a more high-level load balancer that could decide on its own when to send jobs to device with an unified kernel, and when to to a dedicated device with highly optimized execution code that fits most to the device architecture.

5. RELATED WORK

The design space of DBMSs using GPU as co-processor is well explored [5, 26] and already applied in many applications [4, 10, 13, 22]. He et al. present a novel design and implementation of relational join algorithms for GPUs by introducing a set of data-parallel primitives [14]. Stream processing could benefit from this research results in context of DBMSs, but first progress is made. Karnagel et al. show that a stream band-join might be computed faster with a speedup of nearly 70x using a graphic card [18]. For Complex Event Processing, an approach similar to stream processing, Cugola et al. suggest in 2012 to run the pattern detection automaton on parallelized hardware. They conclude that GPU acceleration can bring speedups in this context but also highlight limitations due to memory restrictions [12]. Hence, current approaches for GPU accelerated stream processing focus on specific topics; instead, we suggest an approach to enable GPU-ready stream processing in general. While we focus on a strategy to handle variable-length windows to enable GPU-operators over fixed-sized batches with a column-orientation ("buckets"), Karnagel et al. use a load balancer that mainly deploys band-join computation tasks to both CPU and GPU. Although these tasks also contain tuple-batches from the input sources, our approach has another focus since we do not actually address load balancing and construct batches outside the responsibility of a specific operator or balancer. Hence, we provide a stream of buckets, built from a stream of windows, that can be consumed by any operator. Bucket streams can be shared by different operators and can form an operator chain before the buckets are converted back to a regular tuple stream.

To enable GPU processing capabilities it is reasonable to process batches, since a GPU might outperform a CPU only if a bulk of data is present at once. Some SPSs do only support a tuple-at-a-time approach [15, 25] such that an internal buffering per operator is required. However, our approach enables those architectures to convert tuple-at-a-time windows to bucket streams. Other SPSs such as Aurora offer batch-processing. Here, each operator stores its output in an output queue that is accessible by subscribers via pointers indicating the current ranges in-use. Aurora cleans up these queues, if a tailed range is not used by any subscriber anymore [1]. Since Aurora manages windowing with output queues, these queues vary as the window content and processing performance of subscriber vary. In contrast, our approach uses a fixed-sized circular buffer and performs window portioning and event representation changes rather than managing window states as Aurora.

6. CONCLUSION

In this paper we motivated and introduced a concept for a dedicated stream processing operator, the *bucketing*-operator, that consumes a stream of length-varying windows and produces a stream of fixed-sized window portions with a column-oriented event representation. We motivated the revertible event representation transposition to match the GPU architecture better, since a GPU uses the SIMD approach and otherwise we would waste memory and increase transfer costs. However, we suggest a strategy to perform *bucketing* using a fixed-sized circular buffer for each attribute of a given schema. This approach is efficient in time and space, since it mainly depends linearly on the actual window content length and could be stored in a predefined sized buffer per-attribute. We ensured here, that data corruption cannot occur using our construction.

Finally, we identified two research questions for processing data streams on modern hardware and scheduling for heterogeneous devices.

7. ACKNOWLEDGMENTS

We thank Bernhard Seeger for fruitful discussions that heavily influenced this work.

8. REFERENCES

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. h. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A data stream management system. In *SIGMOD*, page 666, 2003.

[2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Springer, 2004.

[3] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. IBM infosphere streams for scalable, real-time, intelligent transportation services. SIGMOD'10, pages 1093–1104, NY, USA, 2010. ACM.

[4] S. Breß. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[5] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Exploring the design space of a GPU-aware database architecture. In *GID Workshop @ ADBIS*, pages 225–234. Springer, 2014.

[6] S. Breß, N. Siegmund, M. Heimel, M. Saecker, T. Lauer, L. Bellatreche, and G. Saake. Load-aware inter-co-processor parallelism in database query processing. *DKE*, 2014.

[7] D. Broneske. Adaptive reprogramming for databases on heterogeneous processors. In *SIGMOD/PODS Ph.D. Symposium*. ACM, 2015. to appear.

[8] D. Broneske, S. Breß, M. Heimel, and G. Saake. Toward hardware-sensitive database operations. In *EDBT*, pages 229–234. OpenProceedings.org, 2014.

[9] D. Broneske, S. Breß, and G. Saake. Database scan variants on modern CPUs: A performance study. In *IMDM@VLDB*, LNCS, pages 97–111. Springer, 2014.

[10] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, pages 777–786, 2004.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, pages 668–668, 2003.

[12] G. Cugola and A. Margara. Low latency complex event processing on parallel hardware. *J. Parallel Distrib. Comput.*, 72(2):205–218, Feb. 2012.

[13] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management performance graphics co-processor sorting for large database management. In *SIGMOD*, pages 325–336, 2006.

[14] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.

[15] B. Hoßbach, N. Glombiewski, A. Morgen, and B. Ritter, Franz und Seeger. JEPC: The java event processing connectivity. *Datenbank-Spektrum*, 13(3):167–178, 2013.

[16] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. *DaMoN*, pages 55–62, 2012.

[17] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. The HELLS-join: A heterogeneous stream join for extremely large windows. In *DaMoN*, pages 2:1–2:7, 2013.

[18] T. Karnagel, B. Schlegel, D. Habich, and W. Lehner. Stream join processing on heterogeneous processors. In *BTW Workshops*, pages 17–26, 2013.

[19] H. G. Kim, Y. H. Park, Y. H. Cho, and M. H. Kim. Time-slide window join over data streams. *Journal of Intelligent Information Systems*, 43(2):323–347, 2014.

[20] J. Krämer. *Continuous Queries over Data Streams - Semantics and Implementation*. PhD thesis, Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, 2007.

[21] J. Krämer and B. Seeger. Pipes: a public infrastructure for processing and exploring streams. In *Proceedings of the 2004 ACM SIGMOD*, pages 925–926. ACM, 2004.

[22] A. Meister, S. Breß, and G. Saake. Toward GPU-accelerated database optimization. *Datenbank-Spektrum*, 2015. To appear.

[23] S. Z. Sbz, S. Zdonik, M. Stonebraker, M. Cherniack, U. C. Etintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26, 2003.

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

[25] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD*, pages 147–156, 2014.

[26] Y.-C. Tu, A. Kumar, D. Yu, R. Rui, and R. Wheeler. Data management systems on GPUs: promises and challenges. In *SSDBM*, page 33, 2013.