Single Window Stream Aggregation using Reconfigurable Hardware

Prajith Ramakrishnan Geethakumari, Vincenzo Gulisano, Bo Joel Svensson, Pedro Trancoso†, Ioannis Sourdis
Department of Comp. Science and Engineering
Chalmers Universty of Technology, Gothenburg, Sweden
Email: {ramgee, vincenzo.gulisano, joels, ppedro, sourdis}@chalmers.se

***Abstract*—High throughput and low latency stream aggregation – and stream processing in general – is critical for many emerging applications that analyze massive volumes of continuously produced data on-the-fly, to make real time decisions. In many cases, high speed stream aggregation can be achieved incrementally by computing partial results for multiple windows. However, for particular problems, storing all incoming raw data to a single window before processing is more efficient or even the only option. This paper presents the first FPGA-based single window stream aggregation design. Using Maxeler's dataflow engines (DFEs), up to 8 million tuples-per-second can be processed (1.1 Gbps) offering 1-2 orders of magnitude higher throughput than a state-of-the-art stream processing software system. DFEs have a direct feed of incoming data from the network as well as direct access to off-chip DRAM processing a tuple in less than 4 μsec, 4 orders of magnitude lower latency than software. The proposed approach is able to support challenging queries required in realistic stream processing problems (e.g. holistic functions). Our design offers aggregation for up to 1 million concurrently active keys and handles large windows storing up to 6144 values (24 KB) per key.**

I. Introduction

With the recent technological advances the number of connected devices grows rapidly along with the total amount of data they produce. New emerging applications analyze unbounded streams of such *big data* in various domains (e.g. financial, transportation) to make fast, sophisticated decisions. However, real-time analytics of large data streams require *high processing throughput* to cope with massive volumes of data as well as *low latency* to respond in real-time.

Stream aggregation is one of the most challenging and computationally intensive analysis tasks in stream processing. This can be handled by applying the traditional relational database aggregation semantics to a sliding window of a particular size (Window Size - WS). Such window can then be "slided" by a particular number of elements (Window Advance - WA) as to produce new aggregated values. The result is a stream of aggregated values. Incremental aggregation – using multiple windows or panes ([1], [2], [3]) to compute and store partial results rather than storing all the incoming values – has been employed to improve performance and reduce memory pressure (both capacity and bandwidth). However, for some queries, especially with small WA, incremental aggregation has the opposite effect causing an excessive number of memory accesses that limit performance. This is for instance the case in streaming applications processing geo-tagged data [4], social media data [5] or manufacturing equipment data [6]. In such cases, a single window approach that explicitly stores all the incoming values in a single window before processing is more efficient. More importantly, storing values in a single window is unavoidable when computing some holistic functions; these are functions that have no constant bound on the size required to store a partial result, such as `median` [7].

Stream processing and stream aggregation systems in particular have been implemented on various compute platforms. Multicores and GPU are able to sustain high processing throughput but fall short in delivering low latency [8]. They require redundant memory accesses managed by an operating system to store incoming tuples in DRAM even before processing starts. This, besides the long latency, wastes a significant fraction of valuable memory bandwidth reducing performance. On the contrary, FPGAs provide both high processing throughput and low latency. Customized dataflow engines, which naturally match the stream processing characteristics, can be implemented in reconfigurable hardware achieving high throughput [8]. Furthermore, incoming tuples can be fed to an FPGA through a direct network connection with low latency, avoiding unnecessary DRAM accesses.

So far, current FPGA solutions focus on incremental aggregation approaches using multiple window or pane-based designs [1], [2], [3]. As a consequence, queries that require small WA or use holistic functions have poor performance or are not supported at all. Moreover, most existing FPGA designs do not use external DRAM and therefore support a single key or at most a handful of keys, and small window sizes, which are not practical for many real stream processing problems, such as the ones mentioned before [4], [5], [6].

This work addresses the above limitations describing, to the best of our knowledge, the first FPGA-based design for single window tuple based stream aggregation. A Maxeler N-series card is used for the design of a stream aggregation dataflow engine (DFE) [9]. The DFE is fed with incoming tuples through a direct network connection and provides direct access to DRAM through its own memory controller. The main contributions of this work are the following:

- The first FPGA-based design for single window stream aggregation, which (i) uses a Maxeler's Dataflow Engine and deep pipelining to provide processing throughput of up to 8 million tuples per second (1.1 Gbps), and (ii) has a direct network connection to feed incoming tuples as well as direct access to DRAM offering ultra low end-to-end latency of up to 4 μsec.

† Pedro Trancoso is also affiliated to the Department of Computer Science, University of Cyprus, Nicosia, Cyprus.

- An implementation of the above design able to support multiple challenging realistic streaming queries with (i) holistic and arbitrary user-defined aggregation functions, as well as distributive and algebraic ones; (ii) up to 1 million concurrently active keys; (iii) large window sizes storing up to 6144 values per key.

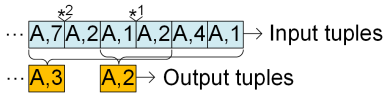
The remainder of this paper is organized as follows. Sections II and III offer background on data stream aggregation and present related work, respectively. Section IV describes the proposed design for single window FPGA-based stream aggregation. Section V presents the evaluation results. Finally, Section VI summarizes our conclusions.

II. Background - Stream Aggregation

In this section, we present the semantics of stream aggregation and provide an overview of the main implementation strategies discussed in the literature. In addition, we elaborate on the two main arguments, discussed in Section I, that motivate our implementation choice among the existing implementation strategies for stream aggregation. More precisely, we count the memory accesses for each implementation strategy, showing that for some problems the algorithm we implement in hardware incurs the lowest DRAM Read/Write (R/W) operations per tuple. In addition, we explain why for some holistic functions explicitly storing all incoming values is necessary.

A stream is an unbounded sequence of tuples $t_0, t_1, \ldots$ each tuple containing n attributes $\langle A_1, \ldots, A_n \rangle$. When aggregated, tuples are fed to one (or multiple) function F such as `sum`, `mean`, `min`, `max` or `count`, among others. Since streams are unbounded, the aggregation is performed over a *sliding window* of *size* WS and *advance* WA. Optionally, aggregation can be performed specifying a parameter K (a subset of the tuple's attributes, also referred to as *key*). In such a case, F is computed for each distinct value observed for K (*group-by*). Figure 1 presents an example in which a stream of tuples with schema $\langle \text{char}, \text{int} \rangle$ are aggregated using parameters F = `mean`, WS = 4, WA = 2 and K = char.

(i)



(ii)

Internal states at time instance	*1	*2
Multi-Window (overlapping windows' counters)	7, 2	5, 2
Pane-Based (Panes' counters)	5, 2	3, 2
Single-Window (tuples' values)	1, 4, 2	2, 1, 2

Fig. 1. Example of stream aggregation over a stream of tuples composed by schema $\langle \text{char}, \text{int} \rangle$ for parameters F=`mean`, WS=4, WA=2 and K=char (i) and internal states maintained by the different implementation strategies at time instance *1 and *2 (ii).

Different strategies exist to implement stream aggregation's semantics. The *Multi-Window* (MW) [1] strategy maintains the partial aggregated state of all the overlapping windows to which each tuple contributes. As shown in Figure 1, the partial state for F = `mean` is the sum of the values observed so far, which is then divided by WS once the window is full. When a window is full, a result is output and the window is discarded. Each tuple contributes to $\lceil \frac{WS}{WA} \rceil$ windows. Upon reception of a tuple, all the overlapping window's states are updated. When the size of the state maintained for each window in DRAM is S, the overall bytes for a key are $\lceil \frac{WS}{WA} \rceil \times S$. Windows are updated upon reception of a tuple incurring $\frac{\lceil \frac{WS}{WA} \rceil \times S}{B}$ R/W from DRAM, where B is the burst size. Furthermore, for every WA tuples, one extra R/W is required to produce the result[1]. The average R/W per tuple and per key is $\frac{1}{WA} + \frac{\lceil \frac{WS}{WA} \rceil \times S}{B}$.

The alternative *Pane-Based* (PB) strategy ([2], [3]), partitions the window into panes of length WA[2] and maintains partial aggregated states for the latter. As shown in Figure 1, the partial state for F=`mean` is the sum of the values observed for each pane of WA tuples. Upon reception of a tuple that fills a window, the result is computed by aggregating its partial states, and the stale panes of the window are then discarded (in the example, summing each pane value and dividing by WS). Finally, the contribution of the tuple is added to the partial aggregate state of the rightmost pane. In this case, when the size of the state maintained for each pane is S, the overall bytes for a key are $\frac{WS}{WA} \times S$. Upon reception of a tuple, updating the last pane incurs one R/W from DRAM[3]. Furthermore, for every WA tuples, $\frac{\lceil \frac{WS}{WA} \rceil \times S}{B}$ R/W are required to produce the result. The average R/W per tuple and per key are then $1 + \frac{1}{WA} \times \frac{\lceil \frac{WS}{WA} \rceil \times S}{B}$.

A third strategy, the *Single-Window* (SW), maintains tuples rather than partially aggregated states. With SW, WS tuples are maintained for each key (as shown in Figure 1) and the results are computed every time the window is full. Being A the size in bytes required to store the attributes to be aggregated, the state is equal to WS × A. In the general access scheme of the DRAM memory, SW requires one R/W for the insertion a tuple and $\frac{WS \times A}{B}$ R/W for the output of a result. Since the latter happens once every WA tuples, the average R/W per tuple and per key are then $1 + \frac{1}{WA} \times \frac{WS \times A}{B}$.

Figure 2 shows the average memory accesses (reads and writes) per tuple incurred by the different strategies when computing the `mean`, the `top-3 highest` and the `top-3 lowest` values of a stream of integer values. For MW and PB, we maintain the sum of the incoming values (as for the example in Figure 1), 3 values for the `top-3 highest` and 3 values for the `top-3 lowest` values, thus incurring a state S of 28 bytes. For SW, A requires 4 bytes. We assume the burst size S is of 64 bytes. As shown, the average number of R/W is lower for SW when several functions aggregate large windows with small advance, as is the case in this work. In the example, the SW approach fits best large windows producing continuous up-to-date results (*i.e.*, with small window advance) by incurring up to 7× fewer memory accesses (for WA=1 and the largest WS) compared to MW and PB (500 for SW instead of 3494 for MW and PB). As explained in Section V-C, even in our SW experiments, performance is limited by

[1]This conservative estimation assumes one window's state fits in a burst.
[2]Assuming WS is a multiple of WA.
[3]Assuming one pane fits in a burst.

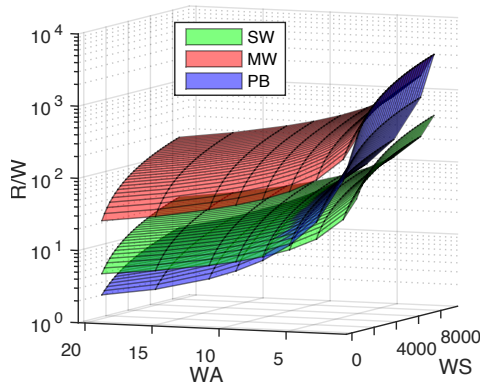


Fig. 2. Average number of memory accesses (R/W) per tuple incurred in MW, PB and SW stream aggregation implementations when computing the average, the top-3 highest, and the top-3 lowest values of a stream of values.

DRAM bandwidth. Then, MW and PB implementations would have a substantially lower performance due to their need for substantially higher DRAM bandwidth. As a consequence, SW is a faster stream aggregation choice for queries with such WA and WS parameters.

Another disadvantage of MW and PB implementation is that for holistic functions, maintaining all input tuples is necessary independently of whether the function can be computed incrementally. This is for instance, the case for the `median` function as all the tuples contributing to the window need to be kept and sorted before the median itself can be computed. This is due to the non-associative nature of the median function. For such functions, a SW implementation is more efficient. MW solutions would require storing all incoming tuples in each one of the multiple windows resulting in many duplicates (exacerbating performance bottleneck) and PB approaches would not be able to maintain partial results.

In summary, for small WA and large WS as well as for particular functions, SW is more efficient for supporting stream aggregation. This motivates our algorithmic choice, the design of which is described in Section IV.

III. Related Work

In this section, we highlight related works that use various computing platforms for stream processing. In addition, we discuss some techniques proposed for in-memory databases (DBs), which are closely related to ours, although they follow a store-and-process paradigm rather than on-the-fly real-time stream processing.

Distributed Stream Processing Engines (SPEs) running on conventional CPUs like Apache Flink, Spark, and Storm provide generic stream processing capabilities and ease of deployment [10], [11], [12]. In particular, Apache Flink is an open-source Java based state-of-the-art stream processing framework. These software-based distributed stream processing engines are easy to configure, flexible to allow for a multitude of operations and analysis on the data, and can process a large amount of data located in different servers. Nevertheless, as with any general-purpose software implementation, their performance depends on the underlying hardware and can never match the throughput or latency offered by dedicated implementations (e.g. custom FPGA-based systems). In the particular focus of this work, software approaches are not able to cope with the challenges posed by aggregating on large windows (large WS) and at high rates (small WA).

SABER is a relational stream processing system targeting heterogeneous machines equipped with CPUs and GPUs [13]. SABER achieves high throughput but at high latency of hundreds of milliseconds for aggregation queries. Moreover, it supports only incremental aggregate computations utilizing the commutative and associative property of some aggregation functions and therefore can implement only distributive (`count`, `sum`) and algebraic (`average`) functions.

Glacier is an FPGA-based streaming query to hardware compiler which supports sliding window aggregation for distributive (`count`, `sum`, `min`, `max`) and algebraic (`average`) aggregation functions [1]. This implementation relies on a MW approach, instantiating multiple aggregation compute modules, resulting in poor scalability in terms of resource utilization and performance. Oge *et al.* improves the aggregation logic in [1] using the PB approach [2], thereby making the design scalable with increasing WS/WA ratio [3]. Nevertheless, both designs use only the on-chip BRAMs for storing aggregation states and do not use DRAM. In turn, this prevents the design from performing stream aggregations of realistic sizes (number of keys, WS). Moreover, Glacier does not support holistic functions such as `median`, as it relies on incremental aggregations. Finally, Oge *et al.* support only a single key and do not support *group-by* clauses in the query [3].

A considerable number of previous works have focused on accelerating in-memory database (DB) operators using FPGAs. For instance, István et al. used an on DRAM hash table [14], which was subsequently improved using a Cuckoo hash table [15], similar to our design. Another interesting work [16], uses an FPGA-based Convey HC-2ex machine to support high throughput *group-by* in-memory DB aggregation, but implements only the `count` aggregation function.

In this work, we propose a single-window stream aggregation implemented on reconfigurable hardware. Our design achieves similar throughput and latency (millions tuples/sec and few μsec, respectively) as other FPGA-based solutions, but is able to produce more complex operations and with larger state. It further confirms prior art conclusions that GPUs have in general much higher stream processing latency than FPGAs [8], as our design achieves 3 orders of magnitude lower latency than related GPU approaches, although none of them has the exact same query semantics considered here. Finally, as shown in Section V, our approach supports substantially higher throughput and lower latency than software.

IV. A Data-Flow Engine for Single Window Stream Aggregation

Data-flow computing matches well the requirements of stream processing. A deep, feed-forward-only pipeline with lightweight control for a back-pressure stall mechanism is able

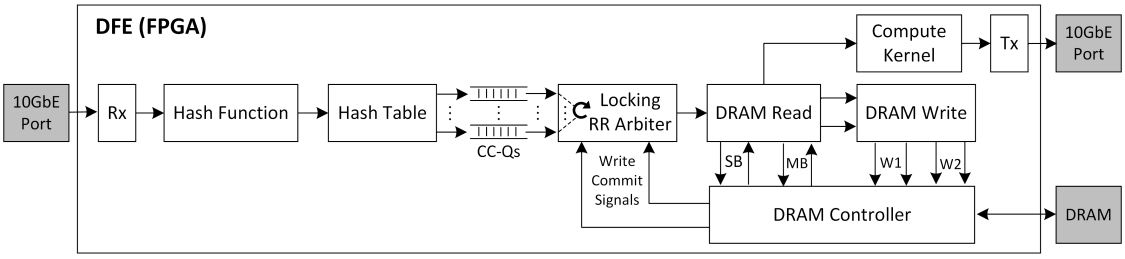


Fig. 3. Top-level view of the Dataflow Engine for single window stream aggregation. The grey boxes indicate peripherals outside the FPGA.

to process fast large volumes of incoming data. We propose a reconfigurable dataflow design for single window stream aggregation. A Maxeler N-series card [9] is used to host our design, implemented as a Data-Flow Engine (DFE), further providing a direct network connection and DRAM access to minimize the processing latency. Although this Maxeler system fits well our design requirements, our approach is general and could be ported to other platforms that exhibit similar characteristics.

Figure 3 shows the top level block diagram of the proposed design. Incoming tuples containing a timestamp, a key, and a value, are carried by network packets and received by the receiver module (Rx). Their keys are hashed to index a hash table, which stores metadata per key, needed for the subsequent processing stages. After accessing the hash table, multiple concurrency control queues are used to enqueue the tuples and resolve dependencies between them. Dequeuing from the queues is performed in a round-robin fashion allowing only for a single tuple per key to be in-flight. Forwarded tuples trigger an access to DRAM to read and subsequently update the stored values (state) of the respective key. When the number of tuples per key reaches the WS threshold, DRAM accesses are issued to read all the values of the corresponding window and compute the aggregation function in the compute stage. The result of the aggregation function is finally transmitted back through the Tx module.

Note, that the flow of the data through the stages is controlled through FIFOs responsible for stalling the pipeline via a back-pressure mechanism when needed as well as for triggering a stage when valid data are available. Bellow, each stage of the stream aggregation engine is described in detail.

A. Receiver and Transmitter

The receiver Rx and transmitter Tx modules handle the incoming and outgoing network packets, respectively, supporting the network protocol processing tasks (TCP, UDP, or Ethernet). Rx and Tx receive and transmit packets from/to the 64-bit wide physical DFE link through protocol specific bi-directional streams. Each packet carries multiple tuples of the following form $\langle ts(8), key(4), value(4)\rangle$, containing in total 16 bytes, of which 8 bytes used for a timestamp, 4 bytes for the key and 4 bytes for the actual value of the tuple.

B. Hash Function

The incoming tuples are unpacked and hashed using their key to generate a hash table address (ht_adr). Bob Jenkin's Lookup3 hash function [17] is used for the hashing as proposed in [14]. Such functions are proven to produce reduced number of collisions due to the *Avalanche* effect, whereby, keys that differ even by a single bit produce different hash values. Two hash functions are computed in parallel, the second one used in case of collision of the first. This stage adds the ht_adr to the tuple and forwards it to the next stage.

C. Hash Table

The hash table, shown in Figure 4, stores metadata for the active keys handled in the system. In order to fit the hash table for a large number of keys using the on-chip BRAMs, our current implementation is direct mapped, having each entry corresponding to a single key. Alternatively, the handling of hash collisions could be improved by adding associativity to the hash table as described in [14]. In such setup, a table entry would offer multiple locations to store a key and a least recently used (LRU) policy would be used for replacement.

As shown in Figure 4, each hash table entry contains (i) a counter for the number of values (tuples) stored in DRAM for a particular key window as to determine when the key context is ready for aggregation, and (ii) a pointer to the last stored value in DRAM (*head pointer*). Optionally, a timestamp or other fields could be added as the application demands it.

As explained in Section IV-E, each entry in the hash table has a statically allocated DRAM memory region to store the values of the respective key. Static allocation simplifies considerably the hardware design compared to a dynamic memory management scheme. The size of this DRAM region determines the maximum window size supported. Due to this static DRAM allocation, after a hash table access, the DRAM location for accessing the stored values of the corresponding key can be calculated using (i) the hash table address storing the key, and (ii) the head pointer to the last stored value.

In general, a tuple accessing the hash table, finds the DRAM location of its key window and decides whether the computation of the aggregation function is triggered. After reading the hash table, the same hash table entry is updated with a new counter-value and head-pointer.

D. Concurrency Control Queues

After the hash table stage, the tuples are distributed among a set of concurrency control queues (cc_Qs) using the least significant bits of their ht_adr. Tuples that belong to the same key are sent to the same cc_Q. Each queue has a fixed time-slot to send a tuple forward and is selected in a round-robin fashion. The cc_Q arbiter has an additional locking

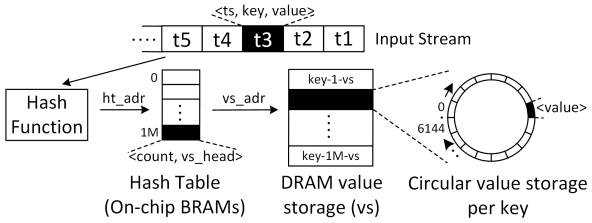


Fig. 4. Hash table and external DRAM organization.

functionality in which it locks the queue as soon as a tuple is pulled out. The arbiter waits for the DRAM write commit signal, indicating that the tuple has updated its key window, before unlocking again the queue. This is done to prevent read-after-write hazards in the pipeline as a result of issuing multiple requests concurrently.

Ideally, blocking of tuples would be avoided (assuming uniform random distribution of keys) when using a number of cc_Qs equal to the delay (in FPGA cycles) of the pipeline part that comes after the concurrency control queues (including DRAM accesses). In practice, we use 128 queues without limiting DFE performance or exhausting FPGA resources.

A similar concurrency control mechanism was proposed for an in-memory DB system in [14]. However, that work uses single registers, rather than entire queues to store tuples. As a consequence, their pipeline is stalled when an incoming tuple needs to be written to an already occupied register. On the contrary, using queues, as proposed in our design, offers higher flexibility and better load balancing, resulting in fewer stalls.

E. DRAM access

The values of each active key, forming a single window, are stored in the external DRAM accessed directly through a memory controller module in the FPGA. Next, we describe the placement of the key values in DRAM as well as the way DRAM reads and writes are performed.

1) Placement of key values in DRAM: The DRAM capacity is equally divided statically to the hash table entries, corresponding to the total number of (active) keys supported. For example, in our platform the total DRAM capacity is 24GB and considering that the total active keys supported is 1 million, each key is allocated a maximum single window value storage of 24 KB. We use the entire DRAM as a single monolithic block in single channel mode with a random access pattern and a single DRAM burst of 192 bytes ($single\_burst\_size$). The number of DRAM bursts corresponding to the maximum single window size is $24KB/192 = 128$. So, for values of size 4 bytes ($value\_size$), the maximum single window size per key is $24KB/4B = 6144$.

The values of a key in the single window are organized in a circular buffer fashion as shown in Figure 4. This is possible as we implement the tuple-based class of sliding window stream aggregation in which the window advance is a fixed number of tuples [18]. Consequently, this allows to overwrite the stale entries in the single window in a circular fashion[4]. The WS and WA in the continuous stream query may be parameterized at runtime. The maximum value of WS is the maximum single window size per key. WA can vary from $1 \ldots$ WS. Using the runtime parameterized WS, the $no\_of\_bursts\_per\_key$ can be calculated as WS $\times$ $value\_size/single\_burst\_size$. For example, if the query has a WS of 3072, then the number of bursts required per key is 64. The granularity of a data word in the DRAM read/write data stream is a single DRAM burst of 192 bytes.

2) DRAM Read: As a tuple along with its control bits enters the DRAM stage, it is known whether an aggregation computation is triggered, or otherwise a new value should just be added to the window. In the former case, a multiple-burst read is issued with start address as vs_adr. In the latter case, a single read is issued at address $vs\_adr + last\_vs\_block$ ($last\_vs\_block$ is retrieved from the *head pointer*). As there are two read streams corresponding to single and multiple bursts, there are two internal queues to buffer the input data words. Once the read is issued, the tuple is placed in a queue, waiting until the memory controller returns with a response.

When DRAM responds to a single read, the tuple awaiting is dequeued and the word read from the DRAM is updated with the new key value and forwarded to the DRAM write stage along with the proper control bits specifying the DRAM location to write back the updated data. Using the above internal queue to delay the tuples with pending DRAM responses, synchronization is achieved between the incoming tuple and its values read from DRAM.

When DRAM responds to a multiple burst read request, a counter with maximum value equal to the burst length ($no\_of\_bursts\_per\_key$) is used to ensure that all the values needed for aggregation are received. Subsequently, the values are forwarded to the compute stage together with the tuple. Note that the new key value carried by the tuple that triggered the aggregation still needs to be written back to DRAM updating the key values. Consequently, a write back to DRAM is performed parallel to the computation of the aggregation.

3) DRAM Write: All writes to DRAM are single burst writes. This stage receives two streams coming from the DRAM Read stage, one for tuples that issued a single word DRAM read, and a second for tuples that issued a multiple burst DRAM read. These two streams bring DRAM write requests, which are sent to the memory controller and subsequently forwarded to the DRAM.

F. Compute Stage

The compute stage calculates the implemented aggregation function, using the values read from DRAM, which correspond to the window of the particular key. Depending on the function in the query (distributive, algebraic, holistic), the values may be processed gradually as they arrive from DRAM (e.g. partial aggregation), or otherwise only when all values in the window have been received. For example, algebraic aggregation functions like `average` and distributive functions such as `minimum`, `maximum`, and `sum` can be partially computed on-the-fly for each burst and then the result

[4]For time-based windows, static memory allocation would limit the maximum number of values arriving within a time slot that defines the window. It would further require variable number of values to be updated in a WA.

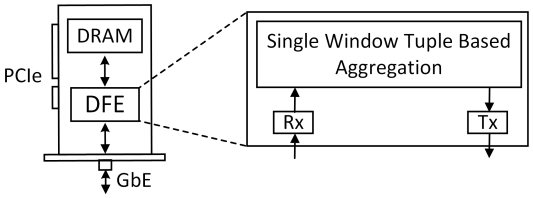


Fig. 5. Maxeler DFE used for SW stream aggregartion.

accumulated to find the final aggregate of the entire window. But for holistic aggregation functions such as `median`, the entire single window should be received before triggering the aggregation. Note that in our implementation of a median aggregation function, the bitonic network sorting algorithm is used, which is a parallel sorting algorithm and conforms to the dataflow paradigm [19]. After the computation of the function is completed the result is forwarded to the Tx stage. Note that the design effort for implementing different queries in our design mainly lies on modifying the compute kernel.

V. Evaluation

The proposed approach is evaluated in terms of performance and power consumption implementing different queries. First, the experimental setup is discussed. Then, the FPGA resource utilization of our design is presented. Finally, processing throughput, latency and power results are reported and compared to a state-of-the-art stream processing software.

A. Experimental Setup

The block diagram of the experimental platform is shown in Figure 5. A Maxeler N-series ISCA (MAX4AB24B) PCIe card with an Altera Stratix V (5SGXAB) was used. The card provides direct 10 GbE network connection to the FPGA. It is further equipped with three 8 GB DDR3 DIMMs accessible directly from the FPGA (24 GB in total). The off-chip DRAM has a maximum bandwidth of 38.4 GB/s and an average latency of about 500 ns. The total on-chip BRAMs available in the DFE is 6.6 MB. Our design is implemented in MaxJ, a Java based High level Synthesis (HLS) language and compiled to FPGA bitstream using MaxCompiler.

Our implementation is compared with the state-of-the-art open source stream processing software framework, Apache Flink (v1.2.1). Flink processing was set up in a workstation with an Intel® Core™ i7-4790 CPU running at 3.6 GHz and 24 GB DDR3 DRAM. Prior FPGA/GPU works cannot be directly compared with our design as they target stream aggregation queries with different semantics.

The data set is uniformly distributed and the tuple's keys and values are generated with uniform distribution. The *tcpreplay* is used to inject captured packets [20]. We experimented with WS up to 6000 and WA ranging from 1 to WS. Our experiments have used a finer grain of WA for the smaller sizes (up to 10) as this is the region of interest for the single window implementation.

The queries which we implemented can be expressed in a streaming relational model [18]. To make the queries intuitive, a subset of the LinearRoad benchmarking system [21] is used, where each vehicle has a sensor that emits tuples composed of a timestamp, a vehicle ID (key), and speed (value).

TABLE I
Resource utilization for the FPGA implementation.

Resource	Query 1	Query 2	Query 3
Logic (ALMs)	89853 (25.01%)	222275 (61.9%)	223364 (62.18%)
BRAMs	1856 (70.30%)	1840 (69.7%)	1906 (72.20%)

The following queries were used in our evaluation:

Query 1: Find the `average`, `minimum`, and `maximum` speed for each vehicle for the last *WS* tuples and return the aggregate every *WA* tuples.

Query 2: Find the `median` speed for each vehicle for the last *WS* tuples and return the aggregate every *WA* tuples.

Query 3: Find the `median`, `average`, `minimum`, and `maximum` speed for each vehicle for the last *WS* tuples and return the aggregate every *WA* tuples.

B. Resource Utilization and Maximum Frequency

The resource utilization of the proposed design is shown in Table I for each query. About 70% of the BRAMs are used for the hash table and the queues employed in our design. It can be further observed that the *median* operation requires more logic resources as it uses a sorting network. The reported resource utilization corresponds to the maximum WS of the queries. For Query 2 and 3, the maximum WS is determined by the largest bitonic sorting network that could fit in the available FPGA resources, which is for a WS of 144[5]. Query 1 operates at 150 MHz. Queries 2 and 3 have more complex implementations and operate at 100MHz.

C. Throughput

The *processing throughput*, *i.e.*, the number of input tuples processed by the system per unit of time, is measured for every query and compared to the software implementation (Flink).

Figure 6 depict the throughput for different Window Sizes (WS), Window Advance (WA) for both the FPGA-based (FPGA), in solid line, and software (Soft), in dashed line, implementations for Query 1. As the WA increases, the throughput also increases. This is expected as the number of aggregates to be produced decreases significantly with the larger WA because the aggregation function is triggered less often. For small values of WA, the software implementation appears to suffer more than the FPGA implementation. This gap closes for larger WA values. The most important observation is that the throughput achieved by the software implementation is always significantly lower by a factor of 13× up to 173× than the FPGA-based implementation, which is able to process 8 million tuples per second.

The charts in Figures 7 and 8 show the throughput for WS of 48, 96, and 144 for Queries 2 and 3 respectively, for both the FPGA and software implementations. For these queries, the

[5]An alternative sorting algorithm such as a merge-sort could have been used but it would still have a storage limitation as all values need to be kept in the FPGA memory to produce the final result.

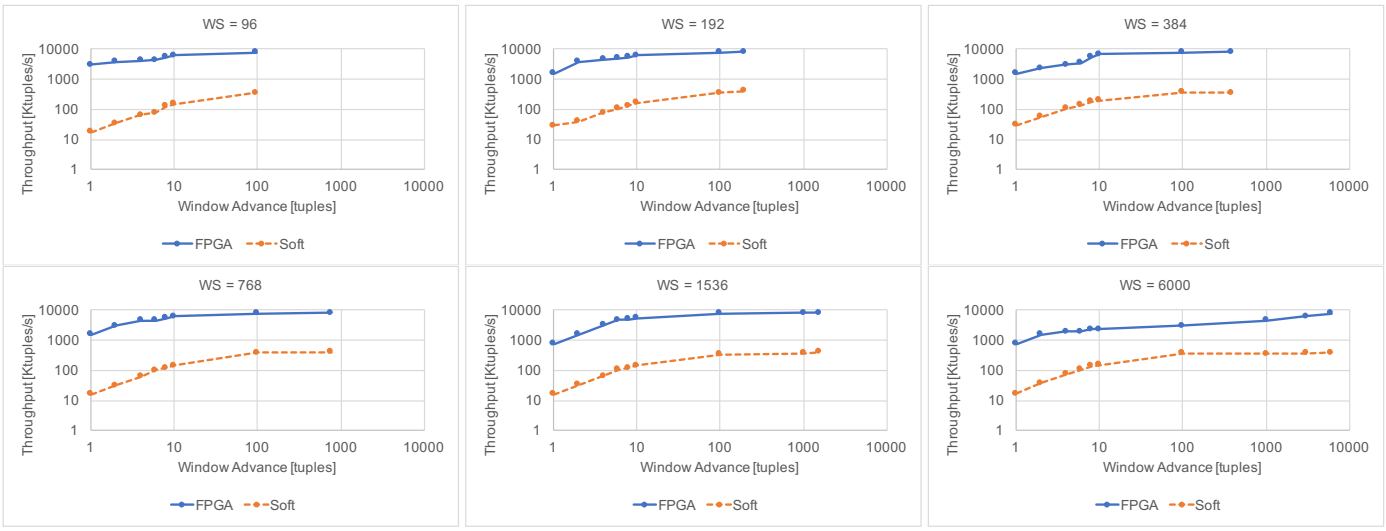


Fig. 6. Throughput for Query 1 for different Window Size values: 96, 192, 384, 768, 1536, and 6000.

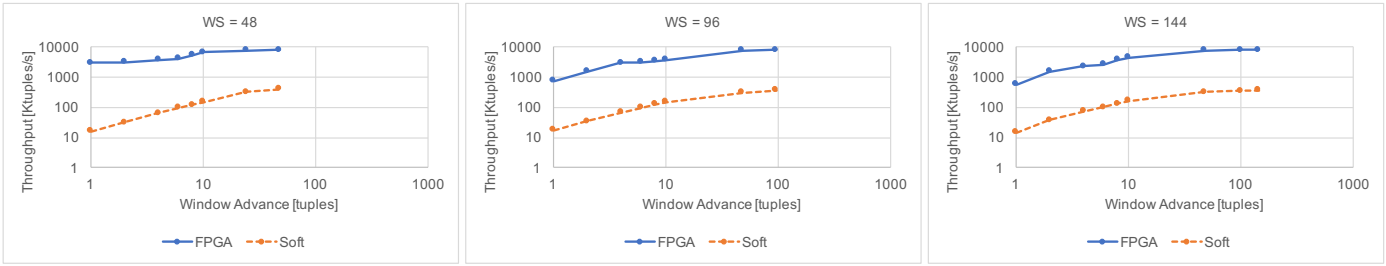


Fig. 7. Throughput for Query 2 for different Window Size values: 48, 96, and 144.

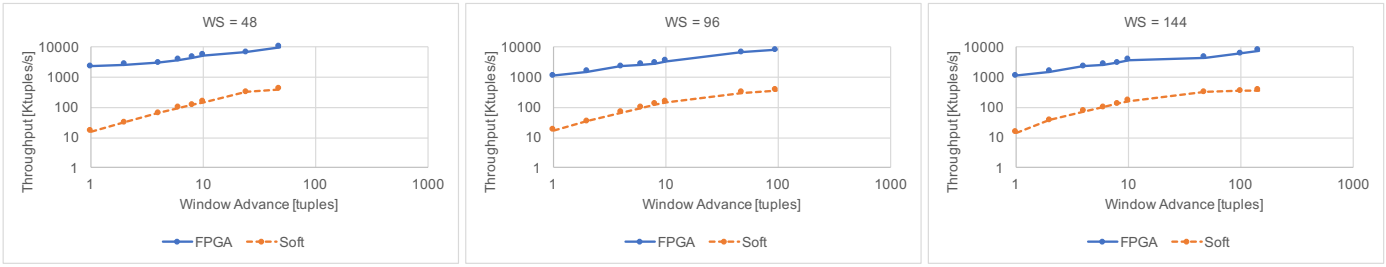


Fig. 8. Throughput for Query 3 for different Window Size values: 48, 96, and 144.

trend is the same as in Query 1: throughput increases as the WA increases, and FPGA throughput is substantially higher than software by a factor of 20× up to 185×.

Overall, stream aggregation in our platform is memory bounded. Considering the operating frequency of our design and the Ethernet bandwidth (10Gbps), DFE would be able to process about 80 Mtuples/sec if not limited by the external DRAM bandwidth. However, the throughput observed in our experiments is an order of magnitude lower.

D. Latency

For the same set of experiments we also measured the average tuple *latency*. That is the average time spend by a single tuple in the system, from the time it enters until its processing is completed. Obviously, the worst case latency occurs when the compute stage is triggered to produce a result for the aggregation function. The contribution of computing the aggregation function in the worst-case overall latency, which is about 4μs, is 0.67μs, 0.79μs, and 1μs for Queries 1, 2, and 3, respectively. In this Section, we show only the latency for Query 1 since it is the simplest and thus the one where the software-based system exhibits the lowest latency (about 1-3% lower than Queries 2 and 3). For the FPGA-based system, the latency is approximately the same across all queries.

Figure 9 shows average tuple latency for different window sizes of Query 1 implemented in software and FPGA. As opposed to the FPGA implementation, in software, WA has some impact in the average tuple latency. Consequently, for software we report the minimum and maximum (average) tuple latency measured for each WS. While not clear in this chart, the software latency is lower as the WA increases. This is because the aggregation function is triggered less frequently and thus the system load is not as high. For the FPGA system the difference is negligible and thus only a single line is shown.

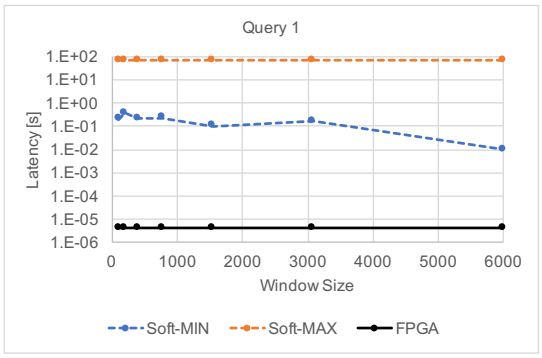


Fig. 9. Latency for Query 1.

The latency for the software system ranges from approximately 0.01 up to 70 seconds. For the FPGA system the latency is significantly lower, approximately 4 μseconds. Even though the operating frequency of the FPGA-based system is much lower (100-150 MHz for the FPGA system as compared to 3.6 GHz for the software) the Maxeler N-series system provides a dedicated FPGA implementation that is substantially more efficient than the general-purpose CPU implementation. In particular, the latency for the FPGA system is 4 to 7 orders of magnitude lower than the latency of the software system.

E. Power

The *power* consumption for both the FPGA-based and software implementations is measured. For the CPU we measured the power consumption using the processor performance counters. This gives package power to which we added DRAM power. For the FPGA we used the available tool to measure the total power consumed by the ISCA board including the FPGA, DRAM and QSFP port.

While for the CPU, power consumption depends on the executed scenario, i.e. the particular query, WS and WA, for the FPGA it is relatively stable. CPU power consumption ranges between 13.7 W to 48.4 W while for the FPGA it is 23.9 W to 29.7 W. Notice that for certain cases the CPU power is actually lower than the FPGA power but for most cases it is almost double.

More relevant than the absolute power consumption is the energy efficiency achieved by each architecture. We define energy efficiency as the Performance (number of tuples processed per second) per Watt. In our experiments, energy efficiency of our FPGA-based design is 10× to 36× better than the software CPU implementation.

VI. Conclusion

High throughput and low latency stream aggregation is critical for various emerging stream processing applications in order to process large data volumes and produce real-time responses. High-end multicores and GPUs are able to support high throughput stream processing, but fall short in delivering low latency. On the contrary, FPGA platforms can provide both. Still, previous FPGA-based solutions offer only incremental stream aggregations, which is not acceptable for certain queries. This work describes the first FPGA-based single window stream aggregation engine. It is able to implement challenging realistic queries of any holistic, distributive or algebraic function and support up to 1 million keys, storing windows of 6144 values per key. Our approach is implemented in a Maxeler N-series dataflow engine, which offers a direct network connection and access to external DRAM. The proposed design achieves 1-2 orders of magnitude higher processing throughput than a state-of-the-art stream processing software system at up to 2× lower power cost, processing up to 8 million packets per second (1.1 Gbps) consuming 23.9-29.7 W. Moreover, a single tuple is processed in less than 4 μsec, at least 4 orders of magnitude faster than software.

Acknowledgment

The authors thank P. Tsigas for his precious comments and feedback. This work was partly funded by the Swedish Research Council under the ScalaNetS project (2016-05231).

References

[1] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: a query compiler for fpgas," *VLDB*, vol. 2, no. 1, pp. 229–240, 2009.
[2] J. Li *et al.*, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *ACM SIGMOD*, vol. 34, no. 1, 2005.
[3] Y. Oge *et al.*, "An efficient and scalable implementation of sliding-window aggregate operator on fpga," in *Int. Symp. on Computing and Networking (CANDAR)*. IEEE, 2013, pp. 112–121.
[4] V. Gulisano *et al.*, "Deterministic real-time analytics of geospatial data streams through scalegate objects," in *ACM Int. Conf. on Distributed Event-Based Systems*, ser. DEBS '15. ACM, 2015, pp. 316–317.
[5] V. Gulisano, Z. Jerzak, S. Voulgaris, and H. Ziekow, "The debs 2016 grand challenge," in *ACM Int. Conf. on Distributed and Event-based Systems*, ser. DEBS '16. ACM, 2016, pp. 289–292.
[6] V. Gulisano *et al.*, "The debs 2017 grand challenge," in *ACM Int. Conf. on Distributed Event-based Systems (DEBS)*, 2017, pp. 271–273.
[7] J. Gray *et al.*, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Min. Knowl. Discov.*, vol. 1, no. 1, pp. 29–53, Jan. 1997.
[8] M. Najafi *et al.*, "Hardware Acceleration Landscape for Distributed Real-time Analytics: Virtues and Limitations," in *ICDCS*, 2017.
[9] "Maxeler-n-series," http://bit.ly/2v1APVK, accessed: October 8, 2019.
[10] "Apache flink," https://flink.apache.org/, accessed: October 8, 2019.
[11] M. Zaharia *et al.*, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM Symp. on Operating Systems Principles*. ACM, 2013, pp. 423–438.
[12] A. Toshniwal *et al.*, "Storm@ twitter," in *ACM SIGMOD Int. Conf. on Management of data*. ACM, 2014, pp. 147–156.
[13] A. Koliousis *et al.*, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Int. Conf. on Management of Data*. ACM, 2016, pp. 555–569.
[14] Z. István, G. Alonso, M. Blott, and K. Vissers, "A hash table for line-rate data processing," *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 2, p. 13, 2015.
[15] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," *VLDB Endowment*, vol. 10, no. 11, 2017.
[16] I. Absalyamov *et al.*, "Fpga-accelerated group-by aggregation using synchronizing caches," in *Int. W. on Data Manag. on New HW*, 2016.
[17] "Bob jenkins lookup3 hash function," http://bit.ly/2vU4cpm, accessed: October 8, 2019.
[18] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.
[19] A. Kos, V. Ranković, and S. Tomažič, "Chapter four-sorting networks on maxeler dataflow supercomputing systems," *Advances in computers*, vol. 96, pp. 139–186, 2015.
[20] "Tcpreplay," http://tcpreplay.appneta.com/, accessed: October 8, 2019.
[21] A. Arasu *et al.*, "Linear road: a stream data management benchmark," in *Int. Conf. on VLDB*, 2004, pp. 480–491.