A Specialized Memory Hierarchy for Stream Aggregation

Prajith Ramakrishnan Geethakumari and Ioannis Sourdis
Department of Computer Science and Engineering
Chalmers University of Technology, Gothenburg, Sweden
Email: {ramgee, sourdis}@chalmers.se

***Abstract*—High throughput stream aggregation is essential for many applications that analyze massive volumes of data. Incoming data need to be stored in a *sliding window* before processing, in case the aggregation functions cannot be computed incrementally. However, this puts tremendous pressure on the memory bandwidth and capacity. GPU and CPU memory management is inefficient for this task as it introduces unnecessary data movement that wastes bandwidth. FPGAs can make more efficient use of their memory but existing approaches employ either only on-chip memory (i.e. SRAM) or only off-chip memory (i.e. DRAM) to store the aggregated values. The high on-chip SRAM bandwidth enables line-rate processing, but only for small problem sizes due to the limited capacity. The larger off-chip DRAM size supports larger problems, but falls short on performance due to lower bandwidth. This paper introduces a specialized memory hierarchy for stream aggregation. It employs multiple memory levels with different characteristics to offer both high bandwidth and capacity. In doing so, larger stream aggregation problems can be supported, i.e. large number of concurrently active keys and large sliding windows, at line-rate performance. A 3-level implementation of the proposed memory hierarchy is used in a reconfigurable stream aggregation dataflow engine (DFE), outperforming existing competing solutions. Compared to designs with only on-chip memory, our approach supports 4 orders of magnitude larger problems. Compared to designs that use only DRAM, our design achieves up to 8× higher throughput.**

I. Introduction

The rapidly increasing rates at which data are produced globally have enabled a large number of emerging stream processing applications [1]. Such applications are employed in various domains, e.g., financial, transportation, to analyze large unbounded streams of data and make fast, sophisticated decisions. However, consuming massive data volumes at line rates requires *high processing throughput* and in case real-time response is expected, it also needs *low latency*.

Stream aggregation is one of the most challenging tasks in stream processing. It can be described by applying the traditional relational database aggregation semantics to a sliding window. Such window of size (WS) is updated with incoming elements (values carried by incoming tuples). Upon aggregation, the window "slides" by a particular number of elements (Window Advance - WA) to produce the aggregated values; that is, the window contents before sliding [2]. The stream of aggregated values is subsequently fed to one or multiple functions that compute an output every time the window slides. Considering a key-value pair system, incoming tuples carry values of different keys, which are aggregated separately using a separate sliding window per key. This description fits a sliding window stream aggregation (SWAG) that follows a tuple-based window policy, meaning WS and WA are measured in terms of the count of elements; an alternative window policy is the time-based one where the size and slide are defined by time intervals [2]. A typical tuple-based SWAG example is depicted in Figure 4(a).

For some problems, the aggregations can be simplified by computing them incrementally using multiple windows/panes [3]–[5]. However, for many others with *non-associative* aggregation functions which cannot be computed incrementally, e.g., `median` [6], or even if they can, it is more expensive than explicitly storing all incoming values in a *single window*, e.g., for processing geo-tagged data [7], social media data [8] or manufacturing equipment data [9].

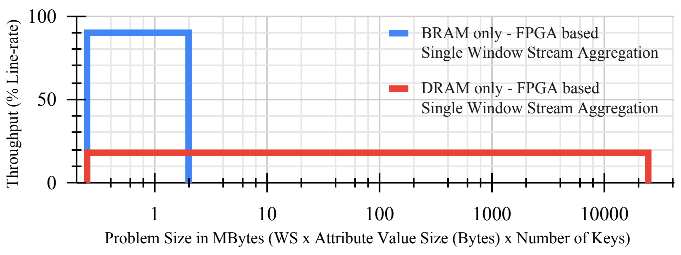


Fig. 1. Processing throughput vs. problem size for an FPGA-based single window SWAG at 156.25 MHz using only BRAMs (2 MB) or only DRAM (24 GB). WS= 2-96K values; WA=WS; 128K keys, value size 2 bytes.

Single window stream aggregation is a memory-intensive problem [10]. For each incoming tuple, the sliding window of the respective key is updated with the newly arrived value(s). In addition, every time the window advances, its entire contents need to be read and fed to the aggregation function(s) to produce a result. Different stream aggregation platforms handle memory in different ways. Multicore CPU and GPU based systems, although able to sustain high processing throughput [11], [12], have wasteful memory management [10]. They require redundant memory accesses to store incoming tuples from the network to DRAM even before processing starts. This, besides the latency overhead, wastes valuable memory bandwidth and hence limits performance. On the contrary, FPGAs use their memory resources more efficiently [13], [14]. They can offer a direct network connection to receive incoming tuples and support dataflow processing, delivering both high processing throughput and low latency. However, existing FPGA approaches use either only on-chip memory (i.e. BRAMs) [3]–[5] or only off-chip memory (i.e. DRAM) to store the values for aggregation [13], [14]. As illustrated in Figure 1 for a particular stream aggregation problem, the higher bandwidth and limited capacity of on-chip BRAM enables line-rate processing on an FPGA based stream aggregation system but supports only small problem sizes. The larger but lower bandwidth off-chip DRAM can handle larger problems, but with limited performance.

This work introduces Multi-level Queues (MLQ), the first memory hierarchy specialized for stream aggregation systems aiming to alleviate their memory bottleneck. The proposed memory system offers a higher and better utilized bandwidth as well as off-chip DRAM capacity to enable higher processing throughput for larger problem sizes, i.e., WS × number of keys. As shown in Figure 2, multiple memory levels are used to form logical queue buffers, each buffer storing the contents of a sliding window for a particular key. Each multi-level logical queue needs to support (i) *single element write* and (ii) *all elements read* operations for window updates and window aggregations, respectively. More precisely, for a window update, a new value needs to be enqueued. The head of the MLQ can be at any memory level, but the tail is always at the fastest (and smaller) first level which is the on-chip SRAM. This ensures that the window is always updated at the highest speed. Then, when the window advances, the contents of the entire window are read utilizing the

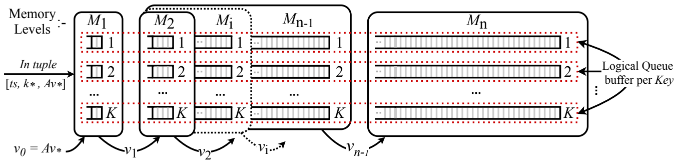


Fig. 2. Memory hierarchical model with n levels with each level having K queues (one per key) and v_i denoting the flushed values from level i. Incoming tuple of key k_* is at timestamp ts with attribute value Av_*.

aggregate bandwidth of all memory levels and subsequently, WA number of elements are discarded by just updating the head pointer. Compared to a BRAM-only design, MLQ offers higher capacity. Compared to a DRAM-only design, it offers faster window updates at on-chip SRAM speed as well as faster aggregation using the aggregate bandwidth of all levels, rather than only the DRAM one.

Another alternative for improving the memory bandwidth of a SWAG could be the use of high bandwidth 3D-stacked DRAM. However, even 3D-stacked DRAM bandwidth is at least an order of magnitude lower than on-chip SRAM, while its size is significantly smaller than off-chip DRAM. In addition, 3D-stacked DRAMs are expensive, especially for systems deployed near the edge. Nevertheless, they could be part of the proposed hierarchy.

Queue buffers composed of two memory types have been designed in the past. More precisely, about two decades ago, 2-level hybrid SRAM/DRAM packet buffers were introduced for network processing [15]–[19] offering SRAM speed and DRAM capacity. Although our approach is in the same direction, there are several fundamental differences. Firstly, the SRAM/DRAM packet buffers implement queues that support read and write operations at the granularity of a single element and this is less bandwidth demanding compared to the stream aggregation. In addition, a network packet size is at least equal to a DRAM line (64Bytes) and therefore fits DRAM better than the stream aggregation accesses which are often finer and hence require expensive read-modify-write operations. Finally, the hybrid packet buffers were limited to two levels, while the proposed memory system can use more levels to exploit a higher aggregate bandwidth.

The contributions of this work are the following: MLQ, a specialized memory hierarchy for stream aggregation; An analytical model of MLQ and a method to automatically generate its configuration for a problem at hand; An MLQ implementation, in a FPGA-based SWAG system and its evaluation and comparison with related work.

The remainder of this paper is organized as follows. Section II offers an analytical model of MLQ. Section III describes our FPGA-based stream aggregation design with MLQs. Section IV presents the evaluation and comparison with related work. Finally, Section V summarizes our conclusions.

II. MLQ Analytical Model

In this section, a generic description of MLQs and an analytical model for estimating system's performance are presented. This model is then used by a heuristic to identify an efficient configuration of the memory hierarchy, i.e., selecting block size per level, for the SWAG problem at hand considering the given memory characteristics while aiming to maximize the processing throughput of the system.

We consider n levels in the memory hierarchy, $M_1, \ldots, M_n$ as shown in Figure 2. The higher the level the longer the access time and the larger its capacity, which is shared equally between the keys (K) supported in the system. Let C_i denote the capacity of the i^{th} memory level and $Ck_i = C_i/K$, denote the capacity available per key in each level. The access granularity of a memory level is defined as the minimum number of bytes that can be read or written (R/W) in a memory access. Let Gw_i and Gr_i denote the write and read access granularity of the i^{th} level, respectively. For example, on-chip SRAM can be configured to support R/W access granularity of just a single bit, while DRAM has a R/W access granularity is 64B lines. The ideal write access granularity (en-queue) required for window updates is equal to size of the attribute value(s) (As) carried by a tuple. In case the write access granularity is larger than that, e.g. values of 4B written directly to DRAM, then a more expensive read-modify-write operation is needed rather than a simple write.

The average R/W memory access time T_i of level i is measured in number of cycles of the processing chip (i.e., FPGA). For simplicity it is considered that all memory ports are R/W with the same access time for both access types. T_i is the inverse of the average access rate (throughput) of a level. The number of available channels offering for independent parallel R/W access in level i is denoted as Ch_i.

We define the input system throughput TPC_{in}, i.e., the line rate at which the system receives incoming tuples. Considering that one tuple is received per cycle, $TPC_{in} = 1$ tuple per cycle.

When a tuple enters the system, its value gets enqueued to the tail of the corresponding key's MLQ in the fastest first level of the memory hierarchy M_1. When a level gets full, the complete block of values in that level is subsequently flushed to the next level. Let F_i denote the flush rate of the i^{th} memory level, which is the inverse of the block of values stored in that level. v_i denotes the number of values stored in the i^{th} level per key before it is flushed to the next level. The incoming tuple's attribute value inserted in M_1 is denoted as $v_0 = 1$. So, $F_i = 1/v_i$, for $1 \leq i \leq n-1$. It is worth noting that the n^{th} level should be able to hold the entire window (v_n = WS).

Based on the above, we model the number of memory accesses per incoming tuple required for window updates and aggregation and then the throughput sustained by each memory level. For simplicity, a uniform random distribution of tuple arrival per key is considered, but different arrival rates would follow the same methodology.

1) Window Updates: There are three types of memory accesses that may occur during window updates. First, a write access of the incoming value(s) of size As to M_1. Second, a read access to any memory level that is full and needs to be flushed. Third, a write access to level i to store the flushed values from level $i-1$. Let Wu_i and Ru_i be the average number of write and read accesses per tuple, respectively, due to window updates to the i^{th} memory level. Every incoming tuple has to be written to M_1, so, $Wu_1 = 1$. For the successive levels, the number of writes in level i depends on the flushing rate per tuple of the previous level F_{i-1} and on the number of write accesses needed to write the v_{i-1} flushed values which is $\lceil \frac{v_{i-1} \times As}{Gw_i} \rceil$. In case a read-modify-write (rmw) operation is needed, an equal amount of read accesses needs to be accounted: $R_{rmw_i} = Wu_i$, if $(v_{i-1} \times As) < Gw_i$; considering values will be aligned and do not span across two M_i lines. Note, that there is the option to underutilize the capacity of memory to avoid read-modify-writes. In the event of a flush, the number of read accesses on level i is $\lceil \frac{v_i \times As}{Gr_i} \rceil$ and on average, these reads happen at a rate of F_i per tuple. Then, the number of write and read accesses for window updates at each level are:

$$Wu_i = \begin{cases} 1, & \text{for } i = 1 \\ F_{i-1} \times \lceil \frac{v_{i-1} \times As}{Gw_i} \rceil, & \text{for } 2 \leq i \leq n \end{cases}$$

$$Ru_i = \begin{cases} F_i \times \lceil \frac{v_i \times As}{Gr_i} \rceil + R_{rmw_i}, & \text{for } 1 \leq i \leq n-1, n > 1 \\ R_{rmw_i}, & \text{for } i = n \end{cases}$$

Algorithm 1: Partitioning algorithm.

```
   Input: $i^1, isBackward$          // mem. level, back-prop. flag
   Output: $V, TPC_{all}$
1  Function Partition($i, isBackward$):
2      if $isBackward$ then
3          if $i == 1$ then
4              update $v_1$, $TPC_1$, $TPC_{in}$
5              Partition($i+1$, false)
6          else
7              update $v_i, TPC_i$ based on $v_{i+1}, TPC_{i+1}$
8              Partition($i-1$, true)
9      else
10         if $hasVisited_i$ then
11             Partition($i+1$, false)
12         else
13             find $v_i$ set to maximise $TPC_i$ within available $Ck_i$
14             if $i == n$ and $v_n < WS$ then
15                 return Not enough capacity in MLQ
16             if $TPC_i == TPC_{i-1}$ and $v_i == WS$ then
17                 print $V$, $TPC_{all}$
18                 return                      // partitioning done
19             if $TPC_i < TPC_{i-1}$ then
20                 Partition($i-1$, true)
21             else
22                 $hasVisited_i$ = true     // visit flag - level $i$
23                 Partition($i+1$, false)
```

[1]Input i encapsulates the memory characteristics of level i such as capacity(Ck), access-granularity (G), and access-time (T).

2) Aggregation: Upon aggregation, the entire window of a key is read, which may spread across all MLQ levels. The number of read accesses in level i for one aggregation is $\lceil \frac{v_i \times As}{Gr_i} \rceil$. Since these reads happen once every WA arriving tuples, then the average number of read accesses per tuple is $Ra_i = \frac{1}{\text{WA}} \times \lceil \frac{v_i \times As}{Gr_i} \rceil$. This is the worst case Ra_i as it assumes the entire key space in the level (v_i) is needed.

3) Average Throughput: The average throughput sustained by level i is measured in tuples per cycle, TPC_i. This is the inverse of the average number of cycles per tuple CPT_i required to complete the accesses per tuple in the level. The CPT_i required per tuple for window update writes and reads as well as for aggregation reads is $[(Wu_i + Ru_i + Ra_i)/Ch_i] \times T_i$, where T_i is the access time (in cycles) and Ch_i the number of parallel channels at the level. Thus, the tuples per cycle for level i is $TPC_i = 1/CPT_i$.

The overall system throughput TPC_{all} is the minimum between the input throughput (TPC_{in}) and the throughput of each memory level. The average cycles per tuple consumed per level are not summed as all memory levels are working in parallel to perform the accesses for window updates and aggregation in a dataflow fashion. So, the throughput supported by the system $TPC_{all} = min(TPC_{in}, TPC_1, \ldots, TPC_n)$. As we focus on the memory system, we consider that the aggregated values delivered by the memory system can be consumed at the same rate by the subsequent stage which computes the aggregation functions, otherwise the throughput of the compute stage needs to be considered in the above equation.

4) Automatic memory configuration generation: Based on the above performance modeling, we seek the optimal memory configuration, that is the number of values stored per level $V = \{v_1, \ldots v_n\}$ based on the aggregation parameters (number of keys, WS, WA) and memory characteristics in order to maximize the processing throughput of the entire system and meet the memory capacity constraints. We developed a heuristic, described in Algorithm 1, to find the partition set V that maximizes system throughput. Starting from level-1, the heuristic finds for each level i, the set of solutions for v_i that can support its requested input throughput TPC_{i-1} within the available capacity Ck_i (line 13) using the formulas of our analytical model. In case there is a set of v_i solutions that can support 100% of TPC_{i-1}, then the output throughput of the level TPC_i is calculated and given to the level $i+1$ as input (line 23). Otherwise, the heuristic would call the function for the previous $i-1$ level (line 20) to reconsider its solutions for the newly adjusted TPC_{i-1} (lines 2-8). The heuristic exits when it finds a solution for the last memory level that satisfies its given input throughput and capacity constraint (lines 16-18). The output of the heuristic is V and the (possibly adjusted) TPC_{all} that is satisfied by V, which is the maximum input throughput TPC_{in} that can be processed by the system. Compared to exhaustive search which would need to search the entire WS space for each memory level ($O(\text{WS}^n)$), our heuristic has substantially lower complexity of $O(n^2)$ as in the worst case, each recursive call has to traverse and fit only from the current level to level-1.

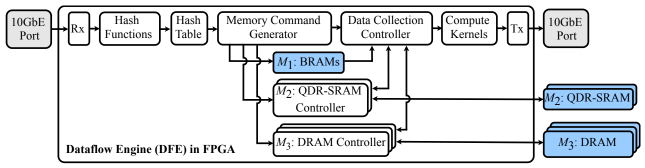


Fig. 3. Top-level view of a Reconfigurable single window SWAG with MLQ.

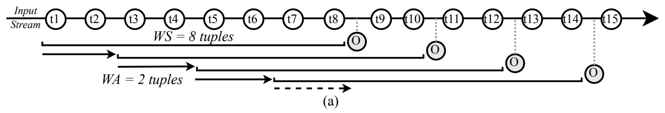
III. Reconfigurable Single Window SWAG with MLQ

A reconfigurable single window SWAG dataflow engine (DFE) that uses the proposed MLQ memory hierarchy is designed to exploit the offered high bandwidth and capacity. The top level block diagram of our engine is shown in Figure 3. Incoming tuples of the form $\langle ts, key, value \rangle$ are carried by network packets and received by the receiver module (Rx).The key of each tuple is first hashed to the hash table. Multiple hash functions are used for reducing collisions and for adding flexibility [20], [21]. Each hash table entry corresponds to a key and stores metadata needed for processing this key's incoming tuples in the subsequent stages; in particular, for managing the memory accesses for window updates and aggregation at each memory level. After accessing the hash table, the memory commands are generated to each level based on the metadata state. The data collector acts as a buffer to synchronise the dataflow of the single window from the various memory levels. Subsequently, aggregated values of a window fetched from the memory are fed to the compute kernel(s) where the aggregation function(s) are computed. The result of the aggregation function is finally transmitted back to the network through the Tx module. Note that the flow of the data through the stages is controlled through FIFOs which stall the pipeline via a back-pressure mechanism when needed. The design is implemented in a platform that offers three memory types to be used in MLQ, in particular besides the FPGA on-chip BRAMs, an off-chip DRAM, and off-chip SRAM are used. Table I shows their characteristics.

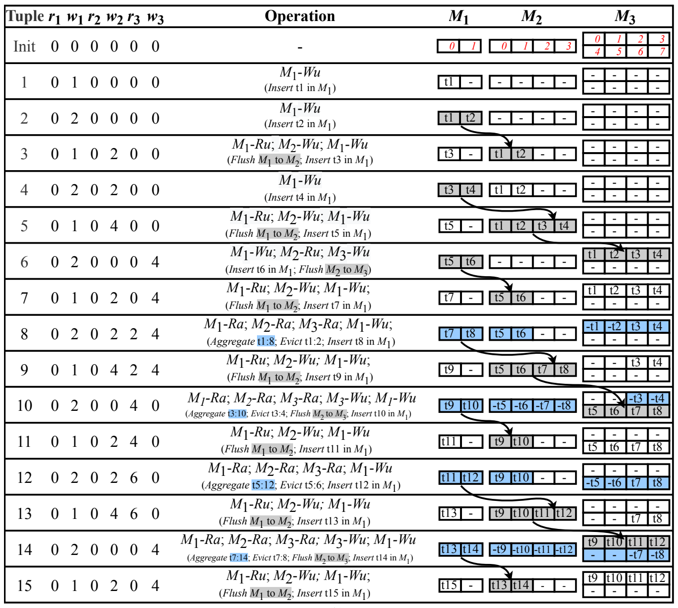
From the end users' perspective, based on the stream aggregation parameters provided and the specifications of the memories, the heuristic discussed in Section II generates the partitioning of the design-time reconfigurable memory hierarchy that maximizes throughput. The APIs provided by MLQ are: a) *insert* (en-queue) a single value upon tuple arrival; b) *flush* a block of values from a full memory level to the next; c) *aggregate* by reading the complete single window queue and d) bulk *evict* values upon window slide corresponding to the WA from the memory levels. These APIs translate to one or more read or write access micro-commands.

A. Hash Table

The hash table stores the metadata for the active keys in the system and is implemented using BRAMs. A hash table entry points to the



(a)



Tuple	r_1	w_1	r_2	w_2	r_3	w_3	Operation	M_1	M_2	M_3
Init	0	0	0	0	0	0	-	0 1	0 1 2 3	0 1 2 3 / 4 5 6 7
1	0	1	0	0	0	0	M_1-*Wu* (*Insert* t1 in M_1)	t1 -	- - - -	- - - - / - - - -
2	0	2	0	0	0	0	M_1-*Wu* (*Insert* t2 in M_1)	t1 t2	- - - -	- - - - / - - - -
3	0	1	0	2	0	0	M_1-*Ru*; M_2-*Wu*; M_1-*Wu* (*Flush* M_1 to M_2; *Insert* t3 in M_1)	t3 -	t1 t2 - -	- - - - / - - - -
4	0	2	0	2	0	0	M_1-*Wu* (*Insert* t4 in M_1)	t3 t4	t1 t2 - -	- - - - / - - - -
5	0	1	0	4	0	0	M_1-*Ru*; M_2-*Wu*; M_1-*Wu* (*Flush* M_1 to M_2; *Insert* t5 in M_1)	t5 -	t1 t2 t3 t4	- - - - / - - - -
6	0	2	0	0	0	4	M_1-*Wu*; M_2-*Ru*; M_3-*Wu* (*Insert* t6 in M_1; *Flush* M_2 to M_3)	t5 t6	- - - -	t1 t2 t3 t4 / - - - -
7	0	1	0	2	0	4	M_1-*Ru*; M_2-*Wu*; M_1-*Wu*; (*Flush* M_1 to M_2; *Insert* t7 in M_1)	t7 -	t5 t6 - -	t1 t2 t3 t4 / - - - -
8	0	2	0	2	2	4	M_1-*Ra*; M_2-*Ra*; M_3-*Ra*; M_1-*Wu*; (*Aggregate* t1:8; *Evict* t1:2; *Insert* t8 in M_1)	t7 t8	t5 t6 - -	-t1 -t2 t3 t4 / - - - -
9	0	1	0	4	2	4	M_1-*Ru*; M_2-*Wu*; M_1-*Wu*; (*Flush* M_1 to M_2; *Insert* t9 in M_1)	t9 -	t5 t6 t7 t8	- - t3 t4 / - - - -
10	0	2	0	0	4	0	M_1-*Ra*; M_2-*Ra*; M_3-*Ra*; M_3-*Wu*; M_1-*Wu* (*Aggregate* t3:10; *Evict* t3:4; *Flush* M_2 to M_3; *Insert* t10 in M_1)	t9 t10	-t5 -t6 -t7 -t8	- - -t3 -t4 / t5 t6 t7 t8
11	0	1	0	2	4	0	M_1-*Ru*; M_2-*Wu*; M_1-*Wu* (*Flush* M_1 to M_2; *Insert* t11 in M_1)	t11 -	t9 t10 - -	- - - - / t5 t6 t7 t8
12	0	2	0	2	6	0	M_1-*Ra*; M_2-*Ra*; M_3-*Ra*; M_1-*Wu* (*Aggregate* t5:12; *Evict* t5:6; *Insert* t12 in M_1)	t11 t12	t9 t10 - -	- - - - / -t5 -t6 t7 t8
13	0	1	0	4	6	0	M_1-*Ru*; M_2-*Wu*; M_1-*Wu* (*Flush* M_1 to M_2; *Insert* t13 in M_1)	t13 -	t9 t10 t11 t12	- - - - / - - t7 t8
14	0	2	0	0	0	4	M_1-*Ra*; M_2-*Ra*; M_3-*Ra*; M_3-*Wu*; M_1-*Wu* (*Aggregate* t7:14; *Evict* t7:8; *Flush* M_2 to M_3; *Insert* t14 in M_1)	t13 t14	-t9 -t10 -t11 -t12	t9 t10 t11 t12 / - - -t7 -t8
15	0	1	0	2	0	4	M_1-*Ru*; M_2-*Wu*; M_1-*Wu*; (*Flush* M_1 to M_2; *Insert* t15 in M_1)	t15 -	t13 t14 - -	t9 t10 t11 t12 / - - - -

(b)

Fig. 4. (a) A stream of tuples t1, t2, …, of a key aggregated with WS=8 and WA=2 tuples. (b) Hash Table metadata logic and dataflow through the memory hierarchy for the above stream. Memories configured with: v_1=2; v_2=4; v_3=8. M_i-Wu, M_i-Ru, and M_i-Ra denotes write operation due to window update, read operation due to window update, and read operation due to aggregation in memory level i, respectively. The red numbers in the $Init$ row indicate the indices of each memory cell. The grey and blue boxes indicate window updates and aggregation reads, respectively.

MLQ space allocated for storing the window of a single key. The metadata contains: (i) a valid bit; (ii) the key assigned to the entry; (iii) window start timestamp for key replacement in case of collision and to trigger aggregation in time-based windows; (iv) read pointers to each memory level, r_i pointing to the head index in level i; and (v) write pointers to each memory level, w_i pointing to the tail index of each memory level i.

The hash table is accessed in a pipelined fashion using multiple hash functions. First, the selected hash table entries are checked to match the requested key. In case of a miss, a new entry is made evicting the least recently used key out of the ones identified by the hash functions. If the evicted key is still active, a flag is raised indicating collision and the information is sent to software.[2]. A hit in the hash table fetches the metadata of the associated key which determine: i) the index in M_1 to *insert* the incoming tuple's value for window update; ii) whether any memory is full and needs to be *flushed* to the next level; iii) the index of the successive level to *insert* the *flushed* block of values from the predecessor; and iv) whether the key is ready for *aggregation* and perform *eviction* of invalid entries on a window-slide. Based on the above cases, the entry is updated and written back to the hash table.

Figure 4 illustrates an example of the index management for tuple-based stream aggregation using the hash table. When the first tuple t1 enters the system, the write pointer w_1 gets incremented by 1 as the tuple is to be inserted in M_1. Similarly, for t2, w_1 becomes 2. On arrival of t3, as $w_1 = 2$ (equal to v_1, the capacity available per key in M_1), M_1 has to be flushed to M_2. Then, t3 is inserted in M1 and w_1 gets updated to 1 pointing to the index in M_1 where the next tuple has to be inserted. Similarly, on arrival of t5, as M_2 is full ($w_2 = v_2$), M_2 is to be flushed to M_3. This goes on until t8 when the total count of tuples in the single window becomes equal to WS ($(w_1 - r_1) + (w_2 - r_2) + (w_3 - r_3) = 8$ tuples) and triggers aggregation. On aggregation, the entire single window spanning across the memory hierarchy has to be read, and then the invalid tuples evicted based on the WA (2 tuples). The eviction is marked by incrementing r_3 by 2 which reduces the total tuple count to 6. An interesting point to note is that on arrival of t10, the buffer in M_3 wraps around and on t14, M_2 gets flushed to the first line in M_3. This maintains the circular buffer per key which is statically allocated in the memory hierarchy. In this example, the read pointers of M_1 and M_2 remain to zero, because on a window-slide (WA=2) the evictions did not span to M_1 or M_2. In a case where the evictions span multiple levels (large WA), the MLQ parts of the upper levels will be emptied and the read pointer of the lower affected level will be updated.

For a tuple-based windowing policy, the read and write pointers memory can be reused to manage evictions. For a time-based case, more metadata are required to manage the number of evicted tuples upon aggregation as described in [14], because the number of tuples per slide (WA) is time-dependent. This is orthogonal to the metadata used for supporting MLQ. Another note is that the MLQ storage in each level $i < n$ can be reduced from v_i to $v_i - v_{i-1}$ because the last write access of v_{i-1} elements to level i triggers a flush to level $i + 1$ and therefore it can be forwarded without storing it in i.

After the hash table access, the tuple along with its hash address, read/write indices, full flag per memory level, and a "ready for aggregation" flag are pushed to the command generator.

B. Memory Command Generator

The command generator controls the memory levels and converts the window update (insert, flush) and aggregation operations into access commands to each memory level. It converts the received indices to actual physical addresses to each memory level. As the available memory space of each level is statically allocated equally to a block per key, the physical address is created using the key block offset. Then, the address of the memory line(s) to be accessed within the key block is generated based on the received index, the value size, and the number of values to be accessed. In case, multiple memory lines need to be accessed (i.e. due to flushing, aggregation), then multiple commands are generated. Finally, in case of a write access at a granularity smaller than a memory line, multiple commands are generated to implement a read-modify-write. Figure 4 describes for the given example the commands to the different memory levels generated by the memory command generator.

After this stage, the tuple along with the number of lines read based on the read commands generated, the full flag per level as well as the aggregation ready flag are passed to the data collection stage.

C. Data Collection Controller

This module synchronizes the data read from memory levels due to flushes and aggregations each of the two cases using a separate state machine. The flushing state machine checks the full-flag of each memory level and writes the flushed data to the next level. In case of a read-modify-write, the lines are modified before flushing to the next level. The aggregation state machine maintains separate internal queues for the tuple's metadata and for the data flowing in from each memory level. It then synchronises the contents of the different queues to deliver in order the aggregated values to the subsequent

[2]The hash table can be extended using the memory hierarchy and/or a software process could handle keys that do not fit in hardware due to collisions or capacity issues.

TABLE I
PLATFORM MEMORY SPECIFICATIONS

Type	Capacity	R/W Access Granularity (bytes)	Theoretical Bandwidth	Avg. Access Time in FPGA cycles (156.25 MHz)
On-chip BRAM	2 MB	4 in our design	14.4 TB/s	1
Off-chip SRAM	72 MB	R: 18 / W: 1	9.9 x2 = 19.8 GB/s	1.2
Off-chip DRAM	24 GB	64	12.8 x3 = 38.4 GB/s	7 (<4 lines), 2 (>4 lines)

TABLE II
RESOURCE UTILIZATION FOR THE FPGA IMPLEMENTATION.

Resource	DFE(D)	DFE(B+D)	DFE(B+Q+D)
Logic (ALMs)	95998 (26.73%)	98137 (27.32%)	102300 (28.48%)
BRAMs	1429 (54.14%)	1705 (64.57%)	1798 (68.09%)

compute kernel. Value reordering is needed because the aggregation read operations are performed in parallel for all memory levels that contain valid data. This is important for supporting non-commutative aggregation functions like `rank`. Finally, the data are pushed to the compute kernel for computing the functions.

D. Computer Kernels

Depending on the aggregation function in the query, the values can be processed on the fly, gradually as they arrive, e.g. for algebraic (i.e.,`average`) or distributive functions (i.e., `minimum`, `maximum`, and `sum`), or otherwise only when all values have been received, e.g., for holistic functions, such as `median`. In our implementation, the compute kernel is pipelined and for `median` a Histogram-based median filtering is implemented [22]. Our design is implemented for a particular (worst-case) WS and WA, but one can choose dynamically at runtime to use a lower WS and/or a different WA, within the same memory configuration. The selected WS and WA is the same for all keys. Multiple queries can be supported by implementing multiple parallel compute kernels. Should these compute kernels be implemented in a way that supports dynamic partial reconfiguration, then these queries could be updated at runtime. After the function computation, the result is forwarded to the Tx stage.

IV. EVALUATION

The performance of the proposed approach is evaluated in terms of processing throughput and latency. First, the experimental setup is discussed. Then, the implementation and performance results are presented and compared against existing approaches.

A. Experimental Setup

All designs are implemented on a Maxeler N-series ISCA (MAX4AB24B) PCIe card with Altera Stratix V (5SGXAB) that provides a 10 Gb/s direct network connection to the FPGA. As shown in Table I, besides on-chip BRAMs and 24 GB DDR3 DRAM, the board offers off-chip QDR-SRAM memory [23]. The designs are implemented in MaxJ, a Java-based High level Synthesis (HLS) language, and compiled using MaxCompiler.

Three different types of FPGA-based single window SWAG dataflow engines (DFEs) are implemented. A SWAG engine that uses only DDR3 DRAM, denoted as DFE(D), that follows the designs described in the current state-of-the-art [13], [14]. A SWAG engine with BRAM and DRAM, denoted as DFE(B+D) and follows the general principles of 2-level SRAM/DRAM hybrid memories used in network processing [15]–[19]. A SWAG engine with a 3-level MLQ memory system using BRAM, off-chip QDR-SRAM and DRAM, denoted as DFE(B+Q+D) and best captures the principles of our approach, although as explained below for some SWAG problems using fewer levels may suffice. The configuration of each design (partitioning V per level) was generated using the proposed heuristic. It is worth noting that none of the SWAG problems sizes used could be supported by a design that uses only BRAM. Finally, a software baseline is used, implemented in Apache Flink v1.5.1 [24] running on an Intel Core i7-4790 CPU at 3.6 GHz using 24 GB DDR3 DRAM.

As a stream aggregation application, a subset of the LinearRoad benchmark [25] is used, where each vehicle has a sensor that emits tuples composed of a timestamp (24b), a vehicle ID (key, 24b), and speed (value, 16b). The data set is uniformly distributed, and the tuple's keys and values are generated with uniform probability. The *tcpreplay* tool [26] is used to inject the captured packets at varying injection rates to determine the highest sustainable system throughput.

The implemented query, comprising of algebraic, distributive, and holistic aggregation functions, is the following: "Find the `average`, `minimum`, `maximum`, and `median` speed for each vehicle for the last WS tuples and return the aggregate every WA tuples." The WS ranges from 64 to 4K tuples, the WA varies from 1 to WS tuples and the number of vehicles (keys) is 128K which is in line with our goal of supporting a wide range of *group-by* stream aggregation queries with large window sizes, frequent aggregations, generic aggregation functions, and a large number of keys.

B. Implementation Results

The resource utilization of the proposed design is as depicted in Table II. The logic utilization is mainly constituted of the hash table stage and the compute kernel (40%) implementing the aggregation functions. The increase in logic utilization across designs is mainly attributed to the increase in pointer arithmetic for managing the extra levels in the hash table stage, command generator, and data collector. The BRAM utilization also increases with more memory levels as more read/write pointers are required in the hash table. For value storage, 512KB of BRAMs are allocated providing 4B per key for DFE(B+D) and DFE(B+Q+D) designs. The remaining BRAMs are mostly utilized by FIFOs between the pipeline stages.

All DFE designs operate at 156.25 MHz allowing to receive one incoming tuple/cycle using the full 10 Gb/s bandwidth of the network interface. This translates to a theoretical line rate of 156.25 million tuples/sec, but in practice, the actual highest rate of incoming tuples measured on the board is about 140 million tuples/sec, so about 90% of the theoretical. Finally, QDR-SRAM and DDR3 DRAM are clocked at 350 MHz and 800 MHz, respectively.

C. Processing throughput and latency

Figure 5 shows the processing throughput of the alternative SWAG designs for different WS and WA, measured in number of tuples processed per unit of time. It also depicts the average (per aggregation) latency, which is interesting for systems with real time constraints.

A general observation is that the throughput of all designs is reduced for small window advance (WA), especially for large window sizes (WS). This is because smaller WA trigger aggregations more frequently and in addition, the larger windows aggregate more data, hence the problem becomes more bandwidth demanding.

The performance of the reference Flink software confirms the claim that CPU memory management does not suit the considered SWAG requirements as it offers 2-3 orders of magnitude lower throughput and 4-7 orders of magnitude higher latency compared to DFEs.

DFEs exhibit similar latency, as DFE(B+Q+D) has on average 90% and 99% the latency of DFE(D) and DFE(B+D), respectively.

The DRAM-only design, DFE(D), which follows the design principles of [13], [14], achieves the lowest throughput out of the three DFEs, supporting up to 15% of the line rate because it handles inefficiently the window update accesses. More precisely, it requires

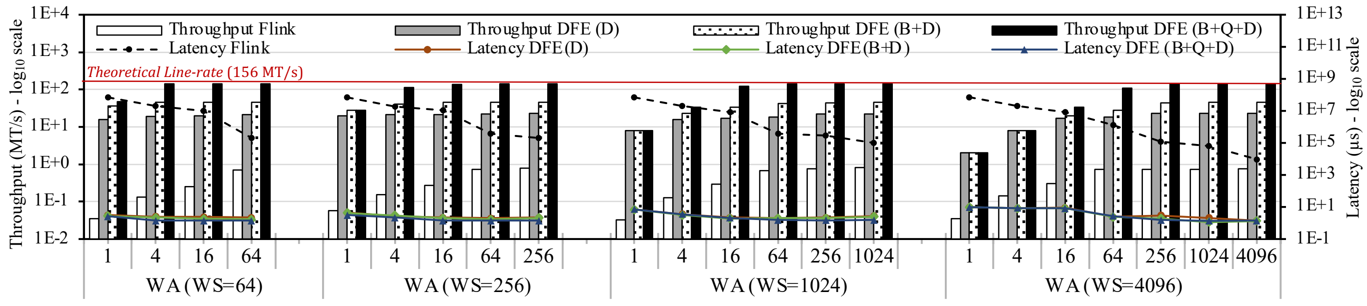


Fig. 5. Throughput and latency of DFEs and Flink.

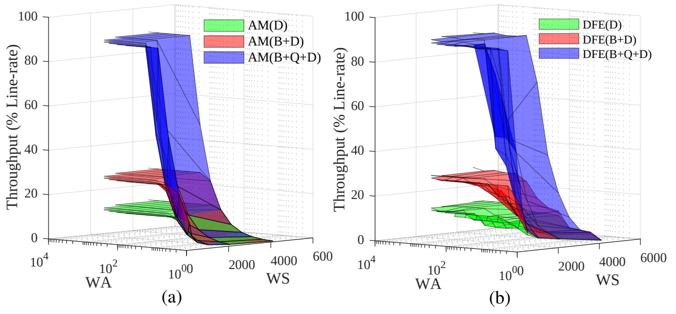


Fig. 6. (a) Throughput estimated by our model. (b) Throughput measured for the actual DFE implementations.

slow and bandwidth wasteful read-modify-writes, since value size (2B) is smaller than DRAM line (64B). However, for small WA (1) and large WS (1024-4096) it offers about the same throughput as the other two DFE designs. This is due to the following reasons. First, in these cases the aggregation traffic dominates and therefore the inefficient handling of window updates has negligible effect. Second, DFE(B+D) and DFE(B+Q+D) cannot take advantage of the added aggregation bandwidth offered by BRAM and BRAM+QDR-SRAM, respectively, because for large WS the dominant portion of the window is stored in DRAM which becomes the bottleneck.

Adding BRAM to form a BRAM+DRAM DFE, DFE(B+D), improves throughput offering up to 30% of the line rate. Although better than DFE(D), the DFE(B+D) memory system is still inefficient. For the given large number of keys, BRAM capacity is too small to fit an entire DRAM line per key, and so cannot completely eliminate read-modify-writes. It can store only two values per key in BRAM, so compared to DFE(D) it reduces the expensive read-modify-write DRAM operations to half, offering better but still limited throughput.

Adding another memory level between BRAM and DRAM solves the above problem and supports up to 90% of the theoretical line rate, which in practice matches the actual maximum rate of incoming tuples on the board. The off-chip QDR-SRAM employed in DFE(B+Q+D) offers the capacity required for storing an entire DRAM line of key-values before flushing to DRAM, completely eliminating read-modify-writes. Moreover, it offers higher aggregate memory bandwidth. This is mostly evident in problems with small WA and WS, because in small WAs, aggregation traffic dominates and in small WS, significant part of the window is not in DRAM, e.g., for WS=64, half of the window is in the lower memory levels.

D. Accuracy of the analytical performance model

Figure 6(a) and (b) show for various WS and WA the system throughput estimated by our analytical model (AM) and measured in our actual implemented DFEs, respectively. The mean absolute percentage error (MAPE) is 12%, but as it can be observed the trend is estimated correctly which is sufficient to guide well our heuristic.

E. Comparison with related work

Compared to existing works, our approach offers higher performance for larger problems due to the use of MLQs.

The DFE(D) implementation follows the design of existing DRAM-only FPGA approaches [13], [14] and is 6-8× slower than our 3-level MLQ design. Another advantage of our approach compared to DRAM-only designs is that it handles skewed key distributions without additional support. The design in [13] would suffer from consecutive tuples of the same key as it would create read after write hazards in the pipeline. The design presented in [14] uses an additional cache structure before the DRAM controller to deal with consecutive DRAM accesses by tuples of the same key. On the contrary, our designs perform all window updates in the fastest level-1 which offers single cycle access and therefore there are no read-after-write-hazards. Moreover, the more recently received values of each key are at the lower levels supporting better performance. We experimented with skewed distribution traffic (from real-world datasets [27], [28]) with the same set of associative and non-associative aggregation functions, and confirmed that the throughput of our design is agnostic to the key-distribution. Finally, BRAM-only FPGA designs such as the ones presented in [3]–[5] can support only 4 orders of magnitude smaller stream aggregation problems; that is up to WS=8 for the considered query and number of keys.

To the best of our knowledge, the only GPU-based stream processing hardware accelerator that supports non-associative functions (i.e., `median`), therefore using a non-incremental aggregation approach is Gasser [12]. However, Gasser supports queries with only a single key, hence small problem sizes, and also does not capture tuples from the network. We experimented with a single key query and DFE(B+Q+D) was able to achieve similar throughput (line rate) comparable to Gasser for varying WS/WA. The throughput readings are similar to the experiment with uniform key distribution traffic as shown in Figure 5. However, DFE achieves this at a much lower latency, which is 3-4 orders of magnitude lower than Gasser.

V. Conclusion

This paper introduced Multi-level Queues (MLQs), a specialized memory hierarchy for stream aggregation. MLQs use multiple memory levels to form logical queues that offer on-chip SRAM (BRAM) bandwidth for window updates and DRAM capacity. In addition, they employ the aggregate bandwidth of all levels in the hierarchy, offering higher aggregation throughput. Compared to BRAM-only stream aggregation designs, MLQ supports 4 orders of magnitude larger problems. Compared to DRAM-only designs, it achieves up to 8× higher throughput. Even compared to hybrid BRAM+DRAM designs, MLQ has up to 4× higher throughput. Finally, MLQ offers the same throughput as GPUs, but in addition supports group-by operations – up to 128K keys rather than one key offered by the competing GPU systems– and 4 orders of magnitude lower aggregation latency.

Acknowledgment

The authors would like to thank Maxeler Technologies for providing infrastructure for running the experiments. This work was partly funded by the Swedish Research Council under the ScalaNetS project (2016-05231).

References

[1] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," December 2012.

[2] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[3] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: a query compiler for fpgas," *VLDB*, vol. 2, no. 1, pp. 229–240, 2009.

[4] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *ACM SIGMOD*, vol. 34, no. 1, 2005.

[5] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "An efficient and scalable implementation of sliding-window aggregate operator on fpga," in *CANDAR*. IEEE, 2013, pp. 112–121.

[6] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Min. Knowl. Dis., 1(1)*, 1997.

[7] V. Gulisano, Y. Nikolakopoulos, I. Walulya, M. Papatriantafilou, and P. Tsigas, "Deterministic real-time analytics of geospatial data streams through scalegate objects," in *ACM DEBS*. ACM, 2015.

[8] V. Gulisano, Z. Jerzak, S. Voulgaris, and H. Ziekow, "The debs 2016 grand challenge," in *ACM DEBS*. ACM, 2016, pp. 289–292.

[9] V. Gulisano, Z. Jerzak, R. Katerinenko, M. Strohbach, and H. Ziekow, "The debs 2017 grand challenge," in *ACM Int. Conf. on Distributed Event-based Systems (DEBS)*, 2017, pp. 271–273.

[10] M. Najafi, K. Zhang, M. Sadoghi, and H.-A. Jacobsen, "Hardware Acceleration Landscape for Distributed Real-time Analytics: Virtues and Limitations," in *ICDCS*, 2017.

[11] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Int. Conf. on Manag. of Data*, 2016.

[12] T. De Matteis, G. Mencagli, D. De Sensi, M. Torquati, and M. Danelutto, "Gasser: An auto-tunable system for general sliding-window streaming operators on gpus," *IEEE Access*, vol. 7, 2019.

[13] P. R. Geethakumari, V. Gulisano, B. J. Svensson, P. Trancoso, and I. Sourdis, "Single window stream aggregation using reconfigurable hardware," in *Int'l Conf. on FPT*. IEEE, 2017.

[14] P. R. Geethakumari, V. Gulisano, P. Trancoso, and I. Sourdis, "Time-swad: A dataflow engine for time-based single window stream aggregation," in *Int'l Conf. on FPT*, 2019.

[15] S. Iyer, R. R. Kompella, and N. McKeown, "Analysis of a memory architecture for fast packet buffers," in *IEEE W. on High Perf. Switching and Routing*, 2001, pp. 368–373.

[16] J. Garcia, J. Corbal, L. Cerda, and M. Valero, "Design and implementation of high-performance memory systems for future packet buffers," in *IEEE/ACM MICRO*, 2003.

[17] J. Garcia-Vidal, M. March, L. Cerda, J. Corbal, and M. Valero, "A dram/sram memory scheme for fast packet buffers," *IEEE Trans. on Computers*, vol. 55, no. 5, pp. 588–602, 2006.

[18] S. Iyer, R. R. Kompella, and N. McKeown, "Designing packet buffers for router linecards," *IEEE/ACM Trans. Netw.*, vol. 16, no. 3, p. 705–717, Jun. 2008.

[19] A. Mutter, "A novel hybrid sram/dram memory architecture for fast packet buffers," in *ACM/IEEE Symp. on ANCS*, 2009, p. 183–184.

[20] A. Kirsch, M. Mitzenmacher, and G. Varghese, "Hash-based techniques for high-speed packet processing," in *Algorithms for Next Generation Networks*, 2010.

[21] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," *VLDB Endowment*, vol. 10, no. 11, 2017.

[22] S. A. Fahmy, P. Y. Cheung, and W. Luk, "Novel fpga-based implementation of median and weighted median filters for image processing," in *IEEE FPL*, 2005.

[23] "Qdr sram," https://intel.ly/2GxJl6m.

[24] "Apache flink," https://flink.apache.org/.

[25] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *Int. Conf. on VLDB*, 2004, pp. 480–491.

[26] "Tcpreplay," http://tcpreplay.appneta.com/.

[27] J. Wilkes, "Google cluster data," https://github.com/google/cluster-data.

[28] H. Ziekow and Z. Jerzak, "The debs 2014 grand challenge," in *8th ACM DEBS*, vol. 14, 2014.