Data stream statistics over sliding windows: How to summarize 150 Million updates per second on a single node

Grigorios Chrysos[†], Odysseas Papapetrou[‡], Dionisios Pnevmatikatos[†], Apostolos Dollas[†], Minos Garofalakis[†*]
[†]School of Electrical and Computer Engineering, Technical University of Crete, Greece
[‡]Department of Mathematics and Computer Science, Eindhoven University of Technology, Netherlands
[*]ATHENA Research and Innovation Center, Athens, Greece

Abstract—**Traditional data management systems map information using centralized and static data structures. Modern applications need to process in real time datasets much larger than system memory. To achieve this, they use dynamic entities that are updated with streaming input data over a sliding window. For efficient and high performance processing, approximate sketch synopses of input streams have been proposed as effective means for the summarization of streaming data over large sliding windows with probabilistic accuracy guarantees.**

This work presents a system-level solution to accelerate the Exponential Count-Min (ECM) sketch algorithm on reconfigurable technology. Different reconfigurable architectures for the sketch structure that correspond to different cost and performance tradeoffs are presented. We map the proposed system-level ECM sketch architectures to a high-end modern HPC platform to achieve guaranteed and best-effort update rates up to 150 and 180 million tuples per second respectively. We compare the performance of the implemented system against the best optimized multi-thread software alternative and show that our scalable full-system accelerators outperform software solutions by 5-7.5x for Virtex6 devices and in excess of 10x for current Ultrascale devices.

I. Introduction

The requirement to process in real time continuous, high-volume data streams is common in many emerging application environments, such as network monitoring for detection of denial-of-service (DoS) attacks, monitoring market data to guide algorithmic trading, and adaptive online advertising. Unlike conventional data processing algorithms for stored data that can utilize several passes over the data, data-stream processing algorithms often rely on building concise, approximate *sketch synopses* of the input streams in real time. Such sketch structures typically require small space and update time (both significantly sub-linear in the size of the data), and can be used to provide approximate query answers with guarantees on the quality of the approximation. These answers are more than sufficient for typical exploratory analysis of massive data, where the goal is to detect interesting statistical patterns rather than obtain precise answers.

In this work we focus on Exponential Count-Min (ECM) sketches that enable the maintenance of distribution statistics for fast-paced data streams over sliding windows [14], [15]. Supported queries include frequency queries (e.g, how many packets did IP address 141.1.1.2 send in the last 2 seconds?), and inner product queries (e.g. what is the inner product of the distribution vector of two streams over the last ten seconds?). Accordingly, they can be used as the primary data structure to maintain heavy hitters, to estimate entropy of the frequency distribution of a stream, and to estimate the similarity of two streams, over varying-length sliding windows. A single-threaded CPU-based implementation of ECM sketches provides throughput of around 10 Million updates per second on recent hardware (cf., Section V-B).

This impressive throughput, however, is still not sufficient in many cases. As an example, consider network routing: modern network switches support up to 150 Million packets per second. Clearly, keeping even approximate statistics at this rate is extremely challenging, and even multi-core CPUs run into bottlenecks such as locks and memory-bandwidth saturation [3], [12]. Hence, hardware accelerators are under consideration for the problem at hand.

GPUs and FPGAs are widely used for big data and stream management [1], [6], [8], [10], [16], [22]. Compared to GPUs, FPGAs have distinct advantages whenever guaranteed throughput is required, and have lower energy footprint and lower Total Cost of Ownership (TCO) vs. state-of-the-art GPUs and manycore processors [4], [7], [9]. Prototypical examples of FPGA-accelerated applications include high-frequency trading, and network management.[1]

In this work, we consider hardware acceleration of ECM sketches. Our two motivating application areas -that stem from our collaborations with companies- are finance and network monitoring. Since FPGAs are already extensively utilized in these fields, we focus on FPGA accelerators. The implementation of ECM sketches over FPGAs raises two fundamental challenges. First, the developer needs to *pre-allocate all FPGA resources (memory and logic) according to the worst-case complexity of the implemented algorithms*. In the case of ECM sketches, the worst-case complexity is logarithmic to the size of the sliding window, whereas the amortized complexity is constant and low. This means that the FPGA resources will be underutilized, opening a potential for smarter approaches and increased utilization. Second, we need to handle *concurrent accesses (R/W) on the same memory location*, i.e, more than one updates that need to modify the same counter in a single cycle. This second challenge is also relevant to the traditional CPU-based ECM sketch implementation.

[1] See, e.g., the NetFPGA initiative (http://netfpga.org/) which develops line-speed NICs, firewalls, and multiport switches on cheap FPGA hardware.

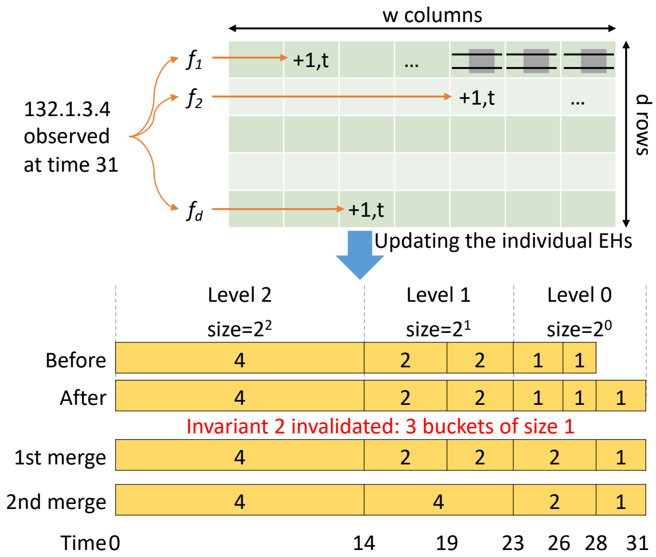

Fig. 1. Updating an ECM sketch.

To address these challenges we examine the design space of the FPGA implementations for the sketch. Each of the proposed implementations in Sections III-IV comes with distinct trade-offs on cost and throughput. We start by discussing a direct mapping of the ECM sketch to reconfigurable hardware, in order to illustrate the basic challenges. Section III-A proposes a fully pipelined and fast mapping with guaranteed throughput, which, however, is too expensive in terms of hardware resources and can only be used with small sliding window sizes. In Section III-B we extend the first architecture by reducing its BRAM memory requirements, and in Section III-C we increase the memory capacity by incorporating DRAM in our architecture so that it can support arbitrarily large sliding window sizes, yet without a drastic reduction of the performance. Finally, Section IV discusses our final architecture which is resilient to skew and supports parallel updates. Section V presents the evaluation the proposed architectures in terms of cost and throughput using traces of real network and financial data as well as two generated traces.

II. ECM SKETCHES

ECM sketch belongs to a family of synopses that allow single-pass summarization of high-dimensional streams over both time-based and count-based sliding windows. The sketch, first introduced in [14], [15], is employed to estimate point (frequency) queries and inner product/join size queries, and it can be employed to address a broad range of problems, such as maintaining frequency statistics, finding heavy hitters, and computing quantiles in the sliding-window model. In the domain of *network monitoring*, which we will be using as a running example throughout the text, the data stream consists of the network packets observed by a router; queries supported by ECM sketch include, e.g., estimating the number of packets sent by any IP address or the number of packets exchanged between a source and a target IP address, and tracking the top-k IP addresses that send the most packets.

The sketch consists of a set of d hash functions $f_1, f_2, \ldots, f_d$, and a 2-dimensional array of counters of width w and depth d (cf., Figure 1). The hash functions map each item from the input domain (i.e., each source IP address) to one counter per row. The counters are modeled as internal data structures that support sliding window queries over binary streams (e.g., exponential histograms or randomized waves). Each stream arrival at time t is handled as follows: first, the key – the source IP address – is hashed by using each of the d hash functions, pointing to one counter per row (e.g., counter 2 at the first row in the example of Figure 1). The corresponding counters are retrieved from the array, and updated by adding one arrival at time t. The updating process of the counters depends on the data structure chosen for implementing the counters. To estimate the frequency of a key – e.g., the number of packets sent by an IP address – we again hash the key using the same d hash functions, to map the key to the d corresponding counters. We then retrieve the counters from the array and query each of them individually, and return the minimum value of all counters as an estimate. In this work we consider *exponential histograms* for maintaining the individual counters, since these are widely used and are more compact than alternatives relying on randomization (e.g., randomized waves), typically by one to two orders of magnitude.

Exponential histograms. Exponential histograms (EH) [5] are deterministic structures proposed to address the basic counting problem, i.e., for counting the number of true bits in the last N arrivals over a bit stream. They operate by breaking the sliding window range into smaller windows, called buckets, to enable efficient maintenance of the statistics. Each bucket contains the aggregate statistics, i.e., number of arrivals and bucket bounds (starting and completion time of the bucket), for the corresponding subrange. Buckets that no longer overlap with the sliding window are expired and discarded from the structure. To compute an aggregate over the whole (or a part of) sliding window, the statistics from all buckets overlapping with the query range are aggregated. For example, for basic counting, aggregation is a summation of the number of true bits in the buckets. A possible estimation error can be introduced due to the oldest bucket inside the query range, which usually has only a partial overlap with the query. Therefore, the maximum possible estimation error is bounded by the size of the last bucket.

To reduce the space requirements, exponential histograms maintain buckets of exponentially increasing sizes. Bucket boundaries are chosen such that the ratio of the size of each bucket b with the sum of the sizes of all buckets more recent than b is upper bounded. In particular, (*invariant 1*) is maintained for all buckets j: $C_j/2(1 + \sum_{i=1}^{j-1} C_i)) \leq \epsilon$, where ϵ denotes the maximum acceptable relative error and C_j denotes the size of bucket j (number of true bits arrived in the bucket range), with bucket 1 being the most recent bucket. (*Invariant 2*) helps us bound the space: the bucket sizes are nondecreasing powers of 2, i.e., $C_i \in \{1, 2, 4, \ldots\}$ and $\forall i : C_i \leq C_{i+1}$, and for every bucket size other than the size of the last bucket, there are at least $\lceil 1/(2\epsilon) \rceil$ and at most $1+\lceil 1/(2\epsilon) \rceil$ buckets. We refer to all buckets with the same size as a *bucket level*. The combination of the two invariants leads to a bound of $\log \frac{2N}{k}$ bucket levels, each containing at most $\frac{k}{2} + 1$ buckets of the same size, with $k = \lceil 1/\epsilon \rceil$. Therefore, each exponential histogram has a total of $O(\frac{1}{\epsilon} \log N)$ buckets, and a space complexity of $O(\frac{1}{\epsilon} \log^2 N)$ – in bits.

To insert an item in the exponential histogram we first create a new bucket of size 1 for the item, add it in the front of the list of buckets, i.e., at position 1, and verify that invariant 2 is valid for buckets of size 1. Whenever invariant 2 is invalidated, i.e., there are more than $1 + \lceil 1/(2\epsilon) \rceil$ buckets of the same size s, the two oldest ones are merged and replaced with a bucket of size $2s$. Notice that a single step of this process may not fully resolve the violation of the invariant, since it may lead to more than $\lceil 1 + 1/(2\epsilon) \rceil$ buckets of size $2s$. In this case, the merging algorithm is applied recursively until it finds a bucket size for which there are at most $\lceil 1 + 1/(2\epsilon) \rceil$ buckets. We call this a *cascading update*.

Example. Figure 1 presents an ECM sketch with $\epsilon = 0.5$. At time $t = 31$, a message from IP address `132.1.3.4` arrives, and is mapped to the corresponding d EHs using $f_1, \dots f_d$. Consider the EH at row d, which is depicted in the figure. After adding the new bucket of size 1 at $t = 31$, we have more than $1 + \lceil 1/(2\epsilon) \rceil = 2$ buckets of size 1. Therefore, the two older buckets of size 1 (the ones that end at time 26 and 28) will be replaced by a new bucket of size 2, which ends at time 28. This will still not fully resolve invariant 2, since we will now have three buckets of size 2. Therefore, the two older buckets of size 2 (the ones ending at time 19 and 23) will be merged to a single bucket of size 4, resolving the invariant.

Configuration of ECM sketches. For values $0 < \epsilon < 1$ and $0 < \delta < 1$, the ECM sketch is configured such that it provides frequency estimates with an error less than ϵN, with probability at least $1 - \delta$ (N denotes the length of the sliding window). Following from Theorem 3 of [15], we configure the ECM sketch by setting the number of rows $d = \lceil \ln 1/\delta \rceil$, the number of columns $w = \lceil e(1 + \epsilon)/\epsilon \rceil$, and parameter k of the individual exponential histograms to $k = \lceil 1/\epsilon \rceil$. This configuration minimizes the space complexity of the sketch.

Update complexity. Due to the possibility of recursive (cascading) updates, the worst-case complexity of updates in exponential histograms is $O(\log N)$, whereas the amortized complexity is constant, with an expected of 2 merges per update. For ECM sketches based on exponential histograms, the worst-case complexity boils down to $O(d \log N)$, whereas the amortized complexity is $O(d)$, with d typically in the range of 3 to 5. Since real-world applications often require large sliding window lengths, the discrepancy between the worst-case and expected cost per update typically exceeds an order of magnitude.

III. FPGA-based Architectures for ECM sketches

Mapping the ECM sketch into hardware is not obvious. The ECM structure, shown at the top of Figure 1, consists of d rows of w EHs, and the size of each EH scales as O(dlogN) (where N is the window size). This makes a fully parallel implementation of $d * w$ independent EHs infeasible. Next, we present three architectures that efficiently map the EH data into reconfigurable logic. The first is a simple, fully pipelined architecture that supports one update per cycle but is costly. The second architecture exploits the nature of the ECM sketches to reduce the required memory resources. The third aims to support larger window sizes via the use of DRAM.

A. Pipelined Architecture: Handling one insertion per cycle

The straightforward fully pipelined (FP) architecture, shown in Figure 2, consists of d parallel modules that map the ECM sketch rows and the corresponding Hash functions. Each ECM row consists of w EHs, and each EH contains L bucket levels. L is determined from the sliding window size and ECM sketch parameters, and grows as $O(log(2N/k) + 1)$, with $k = \lceil 1/\epsilon \rceil$. Updating an EH's buckets on each tuple arrival is a sequential process that lends itself directly to pipelining. Therefore, we model an EH as a linear pipeline of L bucket levels. Exploiting the fact that exactly one EH within each row is updated for every tuple, we group the contents of the w EH of each row into BRAM space using the EH offset as an index. Each time a new tuple arrives at a bucket level, all the corresponding EH values are loaded concurrently to the shift list module from the parallel memory modules. Then, these values are shifted and, finally, they are stored back to the next memory modules. If the update of a specific bucket level creates a new update that needs to pass to the next bucket level (spillover), it is stored in the pipeline register so that it is performed in the next bucket level at the following clock cycle.

The advantages of the proposed architecture are twofold: parallelism, and guaranteed, single cycle throughput. Each of the d ECM rows operate independently, and, within the row, the bucket pipeline prevents the occasional long operation tail to impede processing throughput. Unfortunately, these benefits come at the cost of high (BRAM) resource utilization. Hardware resources are dimensioned for the worst case scenario, i.e. for incoming tuples that update *all* existing levels of the exponential histograms, even though the worst-case scenario is rare. In fact, for each update, the expected number of levels that will need to be updated is 2 (cf. Section II). For the ECM sketch parameters we use in our evaluation, implementing this architecture along with the system peripherals required more than the 1440 BRAMs provided by the Virtex6 FPGA featured in our platform. While this architecture is feasible for moderate to small window sizes and in larger FPGAs, its wastefulness in terms of memory motivated us to consider improvements.

B. Cost-Aware, Full Bandwidth Architecture

To optimize the required resources for the ECM sketch, we exploited two observations: (i) the later parts of the bucket pipeline is rarely used, and (ii) the BRAMs that are used for each bucket are underutilized: only w entries are used and the rest are empty. With w being a relatively small number (55 in our evaluation setup compared to the 512 minimum entries in a BRAM), this leaves a good portion of each BRAM unexploited.

For each of the ECM sketch rows, instead of providing the full -L level deep- bucket pipeline, the proposed cost-aware

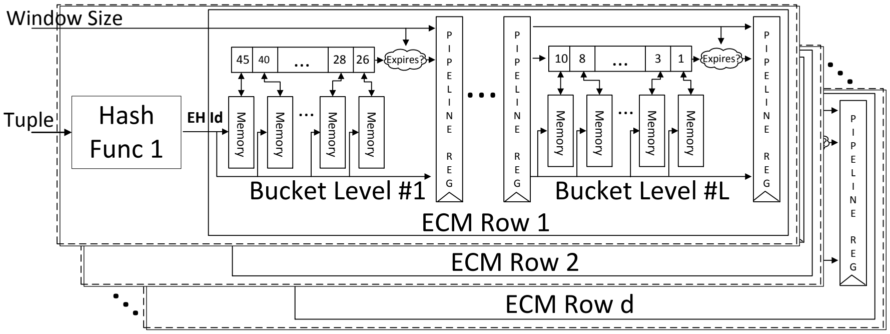

Fig. 2. The Fully Pipelined ECM sketch Architecture groups and maps the EHs of each row ECM structure on parallel reconfigurable components.

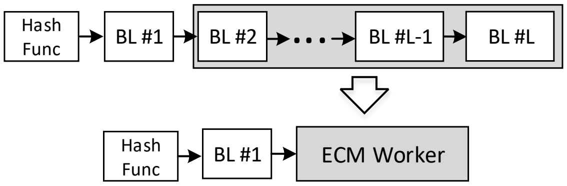

Fig. 3. Cost-Aware ECM sketch architecture: basic idea.

(CA) architecture provides a two stage pipeline. The first stage maps the first bucket level (BL # 1) as in the FP case, while the second stage (called Worker) maps the rest of the bucket levels and corresponding processing. Figure 3 depicts the main idea for this architecture. When a new tuple arrives, it updates the first bucket level of each EH. If this update creates a spillover, it is passed to the Worker stage, which is a single Bucket stage augmented to perform serial bucket processing. If there are further spills, the processing will continue using the loop-back Update FIFO. If new spills arrive from the first stage before the Worker is finished, these are queued in the New Merge FIFO for later processing.

The benefit of this approach is the reduced cost in logic and BRAMs since only two bucket levels are instantiated. However, the processing bandwidth of this architecture is not guaranteed: while an average update touches only two buckets, occasionally it will touch more than two and the Worker module will remain busy for multiple cycles. In addition, merging too many buckets into one does reduce the total BRAMs as they fit all the bucket data. To address these concerns, we instantiate multiple workers per row as shown in Figure 4, essentially over-provisioning the architecture to offer more than average Worker processing bandwidth. This over-provisioning increases logic but not BRAM requirements.

The CA architecture strikes a good balance between resources and performance, but the use of BRAMs for storing bucket data limits the size of the sliding window that can be supported. This is the motivation for the hybrid architecture presented in the following section.

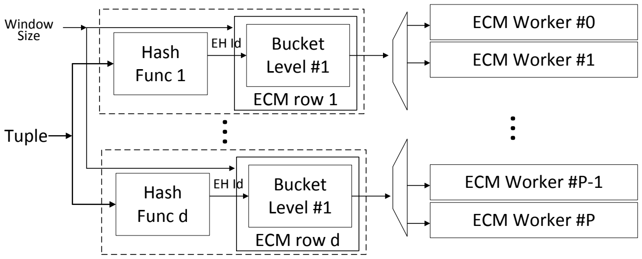

Fig. 4. The full Cost-aware ECM sketch reconfigurable architecture consists of parallel ECM workers for updating data to upper bucket levels.

C. Hybrid Architecture: Supporting Large Window Sizes

To extend the Cost-Aware ECM sketch architecture to support larger window sizes we build on the observation that -on the average- buckets towards the end of the window are rarely updated, so larger window sizes cost in terms of storage but not in terms of processing requirements.

The Hybrid architecture is based on three processing stages (cf., Figure 5). As in the previous case, the first stage is the hashing of the input tuples. Then, and similar to the CA architecture, we have d independent "FrontStage" modules (one per ECM row), followed by a Worker module. Hybrid FrontStage modules implement the first K bucket levels in fully pipelined way, followed by a single (instead of P in CA) Worker module that stores all EH data for the $K+1 \dots L$ bucket levels in DRAM. The large DRAM size allows the Hybrid architecture to support arbitrarily large windows.

K, i.e., the number of the mapped levels, is determined by the available FPGA resources, and should be as high as possible to reduce the load on the DRAM-based BackStage worker. The BackStage handles the spillovers from the K-th level of each row. Its internal structure is similar to the Worker of Figure 3, but it stores all the EH data in DRAM instead of BRAM. If multiple DRAM ports are available to the FPGA, multiple BackStage modules can be instantiated to provide additional backstage processing bandwidth. This additional bandwidth will *not* increase the overall ECM processing throughput; it will only make the system more robust to bursts or long tail tuple arrivals, which is also desirable.

IV. Supporting Multiple Updates per Cycle

Exceeding processing throughput of one tuple per cycle for a single input stream mandates the support of multiple insertions per cycle. To insert T tuples, we need to hash the tuples to determine the $T * d$ EHs that need to be updated. Clearly, this is not possible unless we increase the parallelism

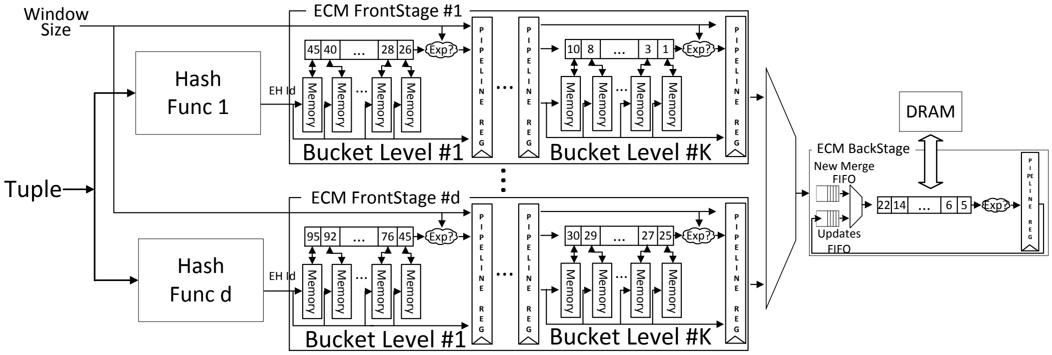

Fig. 5. The Hybrid architecture uses parallel FrontStages and a shared BackStage for updating the bucket levels that are stored in external memory.

in the architecture. Therefore we need more than d FrontStage pipelines, each of which will implement a subset of the ECM sketch row. Assuming that we have multiple FrontStage modules, we rely on the statistical properties of the hash functions to distribute the load evenly.

We start with the DRAM-based architecture of the previous sub-section, and extend it to provide $T*d$ front stage modules. Note that it is not strictly necessary that the number of FrontStage modules is be a multiple of d, but the implementation is simpler if it is. With $T*d$ hash functions we can accept T tuples, and commence their processing in the corresponding FrontStage pipelines. The EHs that correspond to the entire ECM sketch are distributed and mapped into the T FrontStages and the hash functions are adapted accordingly.

Figure 6 presents the proposed Multi-Threaded (MT) architecture. As multiple input tuples arrive, they are hashed in parallel and their EHs IDs are passed to the basic ECM structure. In case that two or more of the incoming tuples collide on an EH, they are enqueued to be served sequentially. Updates to different EHs or different bucket levels of a single EH can proceed concurrently. If the update of the first K bucket levels of the mapped ECM structure leads to a spill-over, it is enqueued to be served by the ECM BackStage module, as described in Section III-C.

While it is easy to see that the design offers the required parallelism to process multiple tuples, we are assuming an even distribution of the load to the FrontStages. In all previous architectures we are guaranteed to update a single EH per row and per tuple. With multiple input tuples, updates to the same EH are possible. To make matters worse, real-world distributions are often highly skewed, i.e., an IP address undertaking a DoS attack will send many more network packets compared to other IP addresses. These packets will lead to updates at the same EHs, and the proposed architecture will need to handle these packets serially, effectively killing the parallelism. Since queuing itself cannot address this challenge, we provide additional *guaranteed* throughput pipelines for a small number of "heavy hitters".

To provide an "escape" path for heavy hitters, FrontStage modules instantiate a parallel pipeline for a *single* EH, using just logic and not BRAMs. A heavy hitter detection block that uses lightweight statistics determines the culprit of a saturation in the input queue. Then, the heavy hitter traffic is split between the main FrontStage pipeline and the additional one. We assume that we can dequeue up to two elements per cycle from the queue and steer them to the correct pipeline.

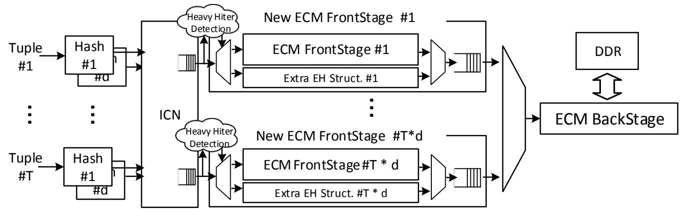

Fig. 6. Multi-tuple per cycle architecture for the ECM sketch; up to P tuples are hashed and directed to the $T*d$ Frontstage modules.

When the system is overloaded with tuples that update the same EH, we split the EH into two sub-EHs, each of which receives and records half of the updates, effectively doubling the processing bandwidth. When the input rate spike subsides, the system disables the escape path and steers future updates to the main pipeline. Queries on this EH are answered combining the data of the two sub-EHs in an addive fashion.The mathematical properties of ECM sketches guarantee that the error bounds (quality) of the answer are not affected.

V. Experimental evaluation

This section evaluates the performance of the proposed reconfigurable architectures for the ECM sketch algorithm.

A. ECM Sketch Setup and Parameters

The ECM sketch data structure size has three parameters, (i) N, the sliding window size, (ii) ϵ, the error factor, and (iii) δ, the probability that the estimation error exceeds ϵ. Unless otherwise mentioned, the window size for our experiments is fixed to two million time units. Typical values for the error rate (ϵ) and the probability value (δ) are between [0.05, 0.25]. We set ϵ = 0.05 and δ = 0.05, as these correspond to the best-quality sketches. The sketches were configured following the discussion in Section II, in order to minimize space complexity and satisfy the chosen ϵ and δ values. For the presented configuration, this led to $w = 55$, $d = 3$, and $k = 11$.

The parameters of our architectures are K, the number of mapped EH levels, and P, the number of parallel Frontstage

Dataset	#Tuples	Update Rate (Million (10^6) Tuples/sec)				
		SWx1/x24	FP	CA	Hybrid	MT
Random1	10^8	10.6/16.4	*150*†	145.1	101.3	178.2
Random2	10^8	10.8/19.9	*150*†	147.3	101.2	177.8
SNMP	$3.1 * 10^7$	11.4/26.6	*150*†	141.1	101.3	173.0
CAIDA	10^8	10.2/19.6	*150*†	147.9	101.2	183.3
WC	10^8	12.2/24.6	*150*†	147.1	101.1	148.5

TABLE I
PERFORMANCE (TUPLE PROCESSING RATE) FOR SOFTWARE (SINGLE AND 24 PARALLEL THREADS) AND SINGLE-FPGA RECONFIGURABLE ARCHITECTURES. † FP PERFORMANCE IS ESTIMATED.

Virtex6 Resources	**FP**	**CA**	**Hybrid**	**MT**
LUTs	137,9K/29%	22,3K/5%	86,3K/18%	223,3K/47%
FFs	57,0K/6%	5,7K/1%	38,5K/4%	141,6K/15%
BRAMs	1071/74%	357/25%	651/45%	847/59%

TABLE II
ACCELERATOR RESOURCE UTILIZATION ON A VIRTEX6 FPGA. NOTE THAT THE COMPLETE SYSTEM USE MORE RESOURCES FOR THE FIXED FUNCTIONALITY, MEMORY CONTROLLER ETC.

modules for the Hybrid architecture and the number of parallel workers for the CA architecture. For the CA architecture P was set to 6, i.e. two parallel workers for each ECM sketch row, due to restrictions in BRAMs for storing the sketch data. For the Hybrid and the MT architectures K was set to 5. This value was determined due to high resource utilization of the FPGA. Last, for the performance evaluation we used T of 3, and a total of 10 FrontStages.

B. Performance and Cost Evaluation

For our experiments we used three real-life trace data sets: (i) the Crawdad SNMP Fall 03/04 [11], (ii) the CAIDA Anonymized Internet Traces 2011[2], and (iii) WC, the data set from world cup98 [2], and two randomly generated traces.

We implemented and evaluated the performance of the four proposed architectures (denoted as *Fully Pipelined* (FP), *Cost-aware* (CA), *Hybrid*, and *Multi-tuple* (MT), respectively), on a Micron (formerly Convey) HC-2ex FPGA-based high-end server that features two six-core Intel Xeon E5-2640 processors, 128 GBytes of main memory, and four Xilinx Virtex-6 LX760 FPGAs with 474240 LUTs, 948480 flip flops, and 1440x18 Kbit BRAMs. The FPGA implementations use only one of the four FPGAs clocked at 150MHz, a frequency set by the HC-2ex system logic.

Performance was measured as the update rate achieved for each of the datasets and is shown in Table I. Software performance was measured using the original single-threaded implementation [15]. We also report the performance of a multi-threaded version of the same code scaling the number of threads up to 24 (the limit of logical processor cores in our system). The parallel software performance saturates at 24 threads and achieves only an overall 2x improvement over single thread due to the high overhead of fine grain locking. The FP implementation could not fit in the FPGA due to BRAM restrictions; we mapped the entire design *without* the HC-2ex fixed interface logic and its post-P&R operating frequency is 160MHz, so we estimate that if it would fit (for a larger FPGA) its operating frequency would be limited by the interface logic at 150MHz. Based on this operating frequency and the fact that it guarantees a processing throughput of one tuple per cycle, we infer that its throughput will be 150Mtuples per second, which is about 15 and 6 times faster than the single- and the multi-threaded software. The CA architecture also achieves close to 150Mtuples per second, loosing some performance due to the occasional Worker overload. The Hybrid architecture, while very flexible, does incur conflicts that bring its performance down to 100Mtuples per second, which still is 5-10x faster than software. The reduced processing throughput is the price for supporting larger window sizes. Finally, the MT architecture is able to sustain processing throughput of 170-180Mtuples per second. The architecture can accept 3 tuples per cycle but the EH conflicts lower the sustained processing rate. Notice that the baseline for comparisons of the MT architecture is the performance of the Hybrid architecture (which also uses DRAM for storage); therefore, the performance improvement by processing multiple tuples is 1.7-1.8x. Also notice that performance varies with the input dataset, with SNMP and WC being the most difficult ones. The effects though are negligible or small across the architectures (the worst effect is for the MT-WC combination).

Table II presents the resource requirements for each accelerator architecture, *excluding* the system logic; in our platform the fixed interfaces, memory controller, etc, use an additional 17% of logic resources and 31% of BRAMs. We can see that CA requires 5 times less logic and 3 times fewer BRAMs compared to FP. We also note that the Hybrid cost is slightly less than that of FP, indicating that supporting larger window sizes via DRAM is certainly feasible. The MT cost in logic compared to that of FP is roughly proportional to the increased throughput it offers.

C. Discussion

The evaluation of the proposed architectures was performed on a Virtex-6 FPGA, and the performance was restricted in the clock frequency by the various system interfaces (memory, bus, etc). Our work can directly benefit from newer devices that are both faster and larger. For example, mid-range Kintex devices provide in the range of 2,000 BRAMs and high-end ones more than 4,000 BRAMs, while Virtex high-end devices provide up to 7,560 BRAMs. To estimate this effect, we mapped our FP logic onto a VCU110 board that features a Virtex UltraScale XCVU190 FPGA. Table III lists the resource utilization and operating frequencies for our proposed architectures. Given the available BRAMs, we can instantiate the entire FP and CA architectures and achieve post place-and-route operating frequencies of 260 and 222 MHz respectively, with corresponding processing rates of about 260 and 214 Mtuples per second. Furthermore, the additional resources can be used to support larger window sizes and/or tighter error bounds (ϵ and δ). The Hybrid and the MT architectures achieve an operating frequency of 244 and 170 MHz, and processing throughput of 165 and 198 Mtuples per second respectively.

[2] Available from http://www.caida.org/data/

UltraScale Resources	**FP**	**CA**	**Hybrid**	**MT**
LUT	62.6K/15%	26.1K/6%	35.8K/8.5%	371.6K/87%
FF	21.5K/2%	8.7K/1%	6.8K/1%	110.4K/13%
BRAM	535/67%	220/28%	168/21%	504/63%
Freq (MHz)	260	222	244	170
Performance (Mtuples/sec)	260	214	165	198

TABLE III
ACCELERATOR RESOURCE UTILIZATION ON AN ULTRASCALE FPGA. NOTE THAT THE COMPLETE SYSTEM WILL USE MORE RESOURCES FOR THE FIXED FUNCTIONALITY, MEMORY CONTROLLER ETC.

Supporting multiple tuples per cycle is challenging: the escape path is an affordable solution but limited both in scope (only one heavy hitter per front-stage module) and in throughput (up to two tuples per second). Borrowing from the CA architecture, we could use multi-ported bucket modules as a FrontStage, and back them with multiple single ported bucket pipelines, striking a reasonable cost-performance trade-off.

Another approach to increase processing rate is to use multiple FPGAs. If the system supports multiple independent ECM sketches, this is relatively straightforward: the incoming tuples will have to be distributed to the corresponding FPGA for processing. However partitioning a single ECM into multiple FPGAs is not straightforward and we expect the synchronization overhead to be significant.

VI. RELATED WORK

Fast streaming data processing is critical in many emerging applications where real-time response is required [24]. Recent works focus on system modeling and design techniques to facilitate streaming applications by exploiting task- and data-level parallelism. Sketch, a highly accurate data stream summarization technique, recently gained much interest in the FPGA research community. SSketch algorithm [18], [19] is the first automated computing framework for FPGA-based online analysis of big data with dense (non-sparse) correlation matrices. It uses streaming input for adaptive learning and updating a corresponding ensemble of lower dimensional data structures. The framework uses a scalable approach for dynamic sketching of massive datasets that works by factorizing the original (densely correlated) large matrix and achieves up to 200x speedup compared to software on a general purpose processor. The SSketch approach is distinct from ECM sketch, thus we can not move on a direct performance comparison.

Saavedra *et al* [20] implemented the Countmin-CU sketch algorithm with the H3 family of hash functions on a reconfigurable platform. The sketch is stored in on-chip memory, and the architecture exploits the parallelism available in the data by simultaneously processing each row of the sketch. Their prototype can process up to a specific number of input elements and achieve twofold performance acceleration. Tong *et al* [23] proposed online based algorithms for two widely used sketches: Count-min and K-ary Sketch. Their implementation focus on 2 key network anomaly detection tasks: heavy hitter detection and heavy change detection. Their system throughput reaches up to 100-150 Gbps for various system configurations. The current work can not directly be compared as the ECM sketch combines two sketches, i.e., CountMin and Exponential Histograms, into a single framework,i.e. ECM sketch. On the other hand, our system offers up to one order of magnitude higher update rates with no restrictions as far as the input size dataset and for much higher complex processing than single CountMin workload.

Sadoghi *et al* [21] presented an efficient multi-query event stream platform that supports only query processing over high-frequency event streams. They used reconfigurable hardware and achieved query processing over 1 Gb/s Ethernet link. Najafi *et al* [13] demonstrated an online reconfigurable event stream query processor, i.e., Flexible Query Processor. Their work focused on addressing performance limitations experienced with general purpose processors needing to operate at line rate using FPGAs. The previous works focused on accelerating query processing over streaming data. Our proposed architectures map both update and query processing over ECM sketch structure with the same performance.

Significant research effort has also been applied to accelerating traditional and in-memory database systems. Papaphilippou *et al* offer a survey of database acceleration frameworks and implementations [17]. While these approaches are very interesting, they focus mainly on the sort, select, join, etc. database operators on datasets that fit in the disk or the main memory, and can be accessed repeatedly to provide exact answers to the queries. As such they are not directly comparable to our work.

VII. CONCLUSIONS

In this work, we explored the potential of FPGAs to efficiently conduct summarization of high-dimensional streams over time-based sliding windows. We mapped a sketch structure, i.e. ECM sketch, on a high-performance reconfigurable platform using different architectures that correspond to different cost-performance trade-offs. Compared to the best optimized state-of-the-art software implementation, our implemented architectures achieve 5-7.5x higher processing rate for both simulated and real data. When we map the proposed architectures to recent UltraScale devices the processing rate reaches 260 Mtuples per second or a 10x speedup compared to software.

While the single-node performance we achieved is significant, it is never sufficient as network speeds and application demands are ever growing. As future work, we intend to further explore the challenges of processing multiple tuples per cycle, exploiting the advantages of the tight coupling between the CPU and FPGA in the Intel Harp platform, and compare with GPU-based mapping of the ECM sketch structure.

ACKNOWLEDGMENTS

This work was supported in part by the European Commission FP7-ICT project Qualimaster (grant agreement #619525), the H2020-FETHPC EXTRA project (grant agreement #671653) and the Marie Sklodowska-Curie MSCA-COFUND-2017 project AQuViDa (grant agreement #665667).

References

[1] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki. The case for heterogeneous HTAP. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[2] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, May/Jun 2000.

[3] L. Azriel, A. Mendelson, and U. C. Weiser. Peripheral memory: A technique for fighting memory bandwidth bottleneck. *Computer Architecture Letters*, 14(1):54–57, 2015.

[4] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 7:1–7:13, Piscataway, NJ, USA, 2016. IEEE Press.

[5] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.

[6] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 3(1):670–680, 2010.

[7] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association.

[8] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 325–336, 2006.

[9] J. Hauswald, M. A. Laurenzano, Y. Zhang, H. Yang, Y. Kang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Designing future warehouse-scale computers for sirius, an end-to-end voice and vision personal assistant. *ACM Trans. Comput. Syst.*, 34(1):2:1–2:32, Apr. 2016.

[10] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 511–524, 2008.

[11] D. Kotz, T. Henderson, I. Abyzov, and J. Yeo. CRAWDAD dataset dartmouth/campus (v. 2009-09-09). Downloaded from https://crawdad.org/dartmouth/campus/20090909, Sept. 2009.

[12] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[13] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on fpgas. *Proceedings of the VLDB Endowment*, 6(12):1310–1313, 2013.

[14] O. Papapetrou, M. N. Garofalakis, and A. Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *PVLDB*, 5(10):992–1003, 2012.

[15] O. Papapetrou, M. N. Garofalakis, and A. Deligiannakis. Sketching distributed sliding-window data streams. *VLDB J.*, 24(3):345–368, 2015.

[16] P. Papaphilippou and W. Luk. Accelerating database systems using fpgas: A survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 125–1255. IEEE, 2018.

[17] P. Papaphilippou and W. Luk. Accelerating database systems using fpgas: A survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 125–1255, Aug 2018.

[18] B. D. Rouhani, A. Mirhoseini, E. M. Songhori, and F. Koushanfar. Automated real-time analysis of streaming big and dense data on reconfigurable platforms. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(1):8, 2016.

[19] B. D. Rouhani, E. M. Songhori, A. Mirhoseini, and F. Koushanfar. Ssketch: An automated framework for streaming sketch-based analysis of big data on fpga. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 187–194. IEEE, 2015.

[20] A. Saavedra, C. Hernández, and M. Figueroa. Heavy-hitter detection using a hardware sketch with the countmin-cu algorithm. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 38–45. IEEE, 2018.

[21] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen. Multi-query stream processing on fpgas. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1229–1232. IEEE, 2012.

[22] D. Sidler, M. Owaida, Z. István, K. Kara, and G. Alonso. doppiodb: A hardware accelerated database. In *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*, page 1, 2017.

[23] D. Tong and V. K. Prasanna. Sketch acceleration on fpga and its applications in network anomaly detection. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):929–942, 2018.

[24] D. Zinn, Q. Hart, T. McPhillips, B. Ludascher, Y. Simmhan, M. Giakkoupis, and V. K. Prasanna. Towards reliable, performant workflows for streaming-applications on cloud platforms. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 235–244, Washington, DC, USA, 2011. IEEE Computer Society.