SKT: A One-Pass Multi-Sketch Data Analytics Accelerator

Monica Chiosa
Systems Group
Dept. of Computer Science
ETH Zurich, Switzerland
monica.chiosa@inf.ethz.ch

Thomas B. Preußer
Accemic Technologies
Dresden, Germany
thomas.preusser@utexas.edu

Gustavo Alonso
Systems Group
Dept. of Computer Science
ETH Zurich, Switzerland
alonso@inf.ethz.ch

ABSTRACT

Data analysts often need to characterize a data stream as a first step to its further processing. Some of the initial insights to be gained include, e.g., the cardinality of the data set and its frequency distribution. Such information is typically extracted by using *sketch* algorithms, now widely employed to process very large data sets in manageable space and in a single pass over the data. Often, analysts need more than one parameter to characterize the stream. However, computing multiple sketches becomes expensive even when using high-end CPUs. Exploiting the increasing adoption of hardware accelerators, this paper proposes *SKT*, an FPGA-based accelerator that can compute several sketches along with basic statistics (average, max, min, etc.) in a single pass over the data. SKT has been designed to characterize a data set by calculating its cardinality, its second frequency moment, and its frequency distribution. The design processes data streams coming either from PCIe or TCP/IP, and it is built to fit emerging cloud service architectures, such as Microsoft's Catapult or Amazon's AQUA. The paper explores the trade-offs of designing sketch algorithms on a spatial architecture and how to combine several sketch algorithms into a single design. The empirical evaluation shows how SKT on an FPGA offers a significant performance gain over high-end, server-class CPUs.

PVLDB Reference Format:
Monica Chiosa, Thomas B. Preußer, and Gustavo Alonso. SKT: A One-Pass Multi-Sketch Data Analytics Accelerator. PVLDB, 14(11): 2369 - 2382, 2021.
doi:10.14778/3476249.3476287

1 INTRODUCTION

Many tasks in data processing, both in streaming scenarios as well as in conventional databases, start with the need to summarize and characterize data sets by computing basic metrics: the cardinality (number of their *distinct* elements), the overall frequency distribution, the heavy-hitters (their high-frequency elements), as well as basic statistics such as average, maximum and minimum values, or histogram distributions of the data. Except for the most basic statistics, there is often a trade-off between computing a given quantity with a certain accuracy and the efficiency of the algorithm used both in time and space. This has given rise to the widespread adoption of sketch algorithms [8, 10, 16] as the canonical approach to compute such metrics over streams and large data sets. For instance, HyperLogLog (HLL) is widely used in systems such as Google's BigQuery to compute the cardinality of data sets with several billion distinct items [25]. Other sketch algorithms are used to, e.g., estimate results for queries of the type `COUNT (DISTINCT _,...)` to predict the size of joins [1] or to compute join orders [34].

When used for data characterization, efficient and useful as they are, sketch algorithms become expensive. First, they require to perform a pass over the entire data set or data stream. Second, summarizing a data stream often involves computing several of its characteristics and not just one. As an example, if a data set has a rather uniform distribution, knowing the frequency of each one of its items is not very informative. Conversely, for a skewed data set, we may want to identify the most common items. Since the data has not been processed before, its characteristics are unknown and, hence, the first step is to compute several such metrics to then decide which ones are useful and provide the most information about the data. In fact, if several metrics are computed, some metrics can be used as quality indicators for the approximations produced by others, an important aspect when using sketches as they produce only approximate results. For this purpose, it would often be beneficial to be able to characterize the data as much as possible in a single pass, for instance, as the data is read from storage.

Unfortunately, computing several sketches over a data set using CPUs is expensive and requires significant CPU capacity. For streams, CPUs are unable to match the bandwidth of a 100 Gbps network, which leads to additional inefficiencies and bottlenecks. This presents a problem for the ever growing amount of data that must be processed under stricter and stricter throughput and latency constraints. The issue is not unique to the use case we present. There is a trend towards hardware specialization that is largely driven by related issues: the complex performance/cost/energy consumption equation in large-scale deployments [24] as well as the need to make data movement more efficient as it is one of the biggest sources of energy consumption in computing infrastructures [14, 15, 27, 45, 48]. Such a trend is now very visible in many deployed and available systems, especially for FPGAs. Microsoft Azure, for example, uses FPGAs embedded on the network data path to off-load network management functionality [18] but also to accelerate a wide range of applications ranging from machine learning [7, 20] to key-value stores [35]. As another example, Amazon's AQUA employs FPGAs together with SSDs to offload parts of SQL to a network-attached caching layer for large-scale data analytics [3], a design also explored in research [28, 49].

To address the problem of efficient data characterization, we take advantage of the growing availability of FPGA accelerators and explore their inherent spatial parallelism to efficiently compute *several sketches and statistics* over a data set *in a single pass*. The resulting system, SKT, provides important insights into the design

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476287

of algorithms for spatial architectures and the use of accelerators for data processing. SKT can be run on a PCIe-attached FPGA card but also on a smart NIC (Network Interface Card) where it processes the data as it arrives from (or departs to) the network. This makes SKT amenable to deployments such as those in Microsoft's Catapult, where the smart NIC approach would allow to add SKT onto any computing node, or in Amazon's AQUA, where SKT could be used to compute key statistics (or even parts of queries) over relational data while it is being sent to the query processing nodes. In this paper, we focus on three complementary sketch algorithms commonly used for characterizing data sets and streams, as well as for cost-based query optimization. These sketches are (a) *HyperLogLog* (HLL), estimating the cardinality or number of *distinct elements* in a data set, (b) *Count-Min*, which estimates *item frequencies*, and (c) *Fast-AGMS*, which estimates the second frequency moment of a distribution. AGMS has been shown to predict the accuracy of a Count-Min sketch when computed over the same data [43]. We will show that the cardinality estimate by HLL also provides another valuable accuracy indication and the three sketches together can be used to obtain a rather accurate picture of the distribution, size, and nature of the data set or stream. Wile these sketches have been been shown to benefit from hardware acceleration [32, 33], SKT is the first system to combine them into a single design and to study the resulting non-trivial resource-accuracy-performance trade-offs.

SKT makes several novel and timely contributions: (1) it explores how to merge several sketches into a single design both on CPUs and FPGAs (Section 3.1), (2) it analyzes in depth the interplay between resource utilization and accuracy of the algorithms (Section 3.4), (3) it addresses several important aspects of algorithm design on spatial architectures (Section 4), (4) it shows how the inherent parallelism of FPGAs can be used to implement combined versions of several sketch algorithms without any of them interfering with each other in terms of performance (Section 5.3), and (5) extensively explores the resulting performance with comparisons against CPUs with different processing capacities (Section 5.2). Our results demonstrate performance gains over CPU deployments: 1.75× over a high-end server with 2× Intel®Xeon®Gold 6248 Processors, and 3.5× over a smaller, more conventional Intel Xeon Gold 6234 Processor. Overall, one FPGA is able to match the performance of 70 cores with the additional advantage that it can process data at line rates as high as 100 Gbps, thereby greatly exceeding what can be done with CPU designs today and significantly contributing to making data processing more energy efficient.

2 BACKGROUND

This section introduces the sketch algorithms used in the paper. An in-depth review of sketches can be found in Cormode et al. [10].

2.1 Sketch Algorithms

Sketch algorithms compute summaries over data streams typically in sub-linear space. Basic scalar summaries include the minimum and maximum values, the stream's size, and its total sum. These four numbers can already characterize the value range, and the average across the whole data stream. More elaborated summaries estimate parameters such as the cardinality or the skew of the data set. There are also sketches that can be queried for item-related estimates, such as individual occurrence counts or item frequencies.

Table 1: Symbols Used for Defining the Frequency Moments

S	Set of data items contained in the stream
f_i	Occurrence count (frequency) of data item i in the stream
F_q	q-th frequency moment

Table 1 lists the symbols used for defining the q-th frequency moments, F_q, of a data stream, a more formal definition of what sketches calculate. They are computed as:

$$F_q = \sum_{i \in S} f_i^q \qquad \text{(using } 0^0 = 0 \text{ for } q = 0\text{)} \tag{1}$$

The zeroth frequency moment, F_0, is the number of *distinct* elements in the stream, i.e., its *cardinality*. The first frequency moment, F_1, is the total number of elements in the stream or *item count*. The frequency moments F_q with $q \geq 2$ represent degrees of skew in the distribution of the data [2]. The second frequency moment, F_2, is particularly relevant in query optimization as it is used to estimate self join sizes [1].

2.2 HyperLogLog (HLL)

The HyperLogLog (HLL) sketch estimates the number of distinct data items in a data stream (its zeroth frequency moment) [19]. Cardinality estimation is commonly used in databases, e.g., for approximate query processing [6], query optimization (DBMS) [38, 53], and data mining [37]. HLL has become the standard algorithm to estimate cardinalities in very large data sets. It is used by Google [25, 26] in BigQuery for data analytics, by Facebook [4] to estimate their graph's degrees of separation for social network analysis, and in systems such as Amazon's Redshift [39] or Databricks [44].

HLL is a hash-based algorithm. It maps each item of the data stream to a hash value and monitors the maximum number of leading zeros observed among the binary representations of these hash values. HLL exploits that a randomized hash with i leading zeros is expected to be seen only once across 2^i distinct elements. Thus, one needs to process around 2^i elements to observe a hash containing i leading zeros. For the same reason, if a maximum of i leading zeros have been seen, one has probably processed 2^i distinct elements. The approach can lead to large errors overestimating the cardinality, for instance, through encountering a value with many leading zeros in a short data stream. To mitigate such anomalies, adjustments are made such as using parallel sub-sketches and correcting the estimated value for particularly small cardinalities.

HLL maintains a one-dimensional, zero-initialized array as its underlying data structure. For an item insertion, part of its hash value indexes into this array to identify an update location. The remaining hash is subjected to a leading-zero count. The resulting number plus one is called the *rank*, $\varrho(\cdot)$. The identified update location is then set to the maximum of its current value and the computed rank. Ultimately, each array element stores the cardinality estimate for the input substream whose hash values produced the corresponding array index. The overall stream cardinality is estimated as the harmonic mean across all these partial estimates

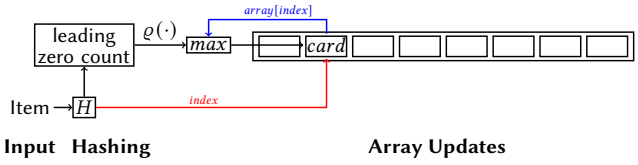


Figure 1: The HLL Update Process

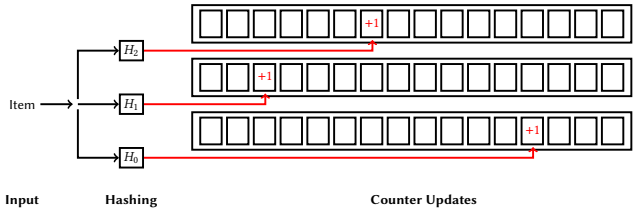


Figure 2: Item Insertion into a Count-Min Sketch

normalized by the number of differentiated indexes. Statistic low-cardinality corrections are applied if some array elements remain untouched at a value of zero. Figure 1 depicts the HLL update process using a very small sketch array that differentiates eight array indexes (aka. buckets) by using three hash bits as index and the rest for leading zero detection.

Hyperloglog has an expected relative error of $\pm\frac{1.04}{\sqrt{2^P}}$ [19], where 2^P represents the number of array elements (P is the precision, see Table 2). This means that a stream's cardinality can be estimated within an error margin of 1-2% with a memory footprint of only a few KBytes [8]. An analysis by Databricks on Spark's HLL performance shows that the error margin can be reduced to values below 1% at the expense of memory usage and speed [44]. When keeping the estimation error below 1%, Spark's HLL algorithm is slower than the actual count of distinct elements [44], proving the significant overhead of computing HLL on conventional CPUs.

2.3 Count-Min

Count-Min [11] computes the approximated occurrence of data items in a data stream. It can be queried for item counts, heavy hitters, or the most popular items in a data stream [52]. It is also used to find the optimal join order for multiple joins [34].

Count-Min is similar to Bloom filters [5]. It differs from them by offering an estimate of the item frequency rather than just a binary set membership. Count-Min is closely related to Counting Bloom Filters as used to maintain a cache summary of a Web proxy [17]. Both algorithms maintain a set of zero-initialized counters. The update locations associated with a data item are identified by a fixed number of hash functions. All counters at the identified locations are incremented for an item insertion and decremented for an item deletion. While a Counting Bloom Filter uses a one dimensional array, Count-Min uses a two-dimensional counter matrix with a fixed association of each row with one of the hash functions. Count-Min has, thus, a structural benefit by performing parallel updates into independent memory regions. We will exploit this feature in the FPGA implementation of Count-Min.

Figure 2 illustrates the insertion of an item into a Count-Min sketch. This small sketch example maintains $R = 3$ rows with 16 (2^P with $P = 4$) counters each. For each inserted data item, independent hash functions H_i derive 4-bit addresses that identify the counters to be incremented in each of the rows. Count-Min supports deletions. The corresponding sketch update process is identical, except for the identified counters being decremented rather than incremented. To find out the frequency of an item, a query follows the same procedure to access the counters. However, it only reads the counter values in order to report the minimum of all values read as the estimated item's frequency. Observe that this result will never underestimate the actual item frequency as every counter included in this minimum has been incremented for every occurrence of the queried item. However, overestimation is possible in the presence of hash collisions with other items. This effect is mitigated by taking the minimum across the R rows. In this way, only the smallest contribution produced by collisions under independent hash functions may affect the reported estimate. The effect of collisions is also significantly reduced since two items would need to collide in all hash functions in order to affect the count of each other.

Count-Min estimates an item's frequency with an error of at most εC with a probabilistic confidence $(1-\delta)$ [10] where $\varepsilon = \frac{2}{2^P}$ with 2^P being the number of counters, $\delta = \frac{1}{2^R}$ with R being the number of rows, and C representing the expected number of colliding items. The value of C can be assessed by correlating the input stream cardinality with the number of counters. We will take advantage of this feature and use HLL to calibrate Count-Min.

2.4 AGMS

AGMS estimates the second frequency moment [2], also known as *the repeat rate or Gini's index of homogeneity*. The sketch itself was first introduced for relational databases in order to estimate the size of joins in the context of limited storage [1]. Traffic validation systems [51] also employ AGMS to identify anomalies between the incoming and outgoing traffic.

Like the Count-Min sketch, AGMS maintains a two-dimensional array of counters where each row has its own independent hash function. When inserting a data item, however, *all* counters in the sketch are updated. Each hash function produces one bit for each column in the row. This bit decides whether each individual update is an increment or a decrement. Deletions can be supported by reversing the sign of these updates. The result is obtained by extracting the median from the arithmetic means of the squared counter values computed for each row.

Cormode and Garofalakis [9] identify the parallel update of the *whole* AGMS counter matrix as a major performance bottleneck. They propose a modified algorithm. Instead of using the hash to choose between incrementing or decrementing each and every counter, they limit the update to a single counter in a row. Both the counter's index and the direction of its update are determined by the hash value of the data item. This modification makes the AGMS update procedure similar to that of Count-Min. The only difference is that the row hash needs one additional bit to select between an increment or decrement for the performed counter update.

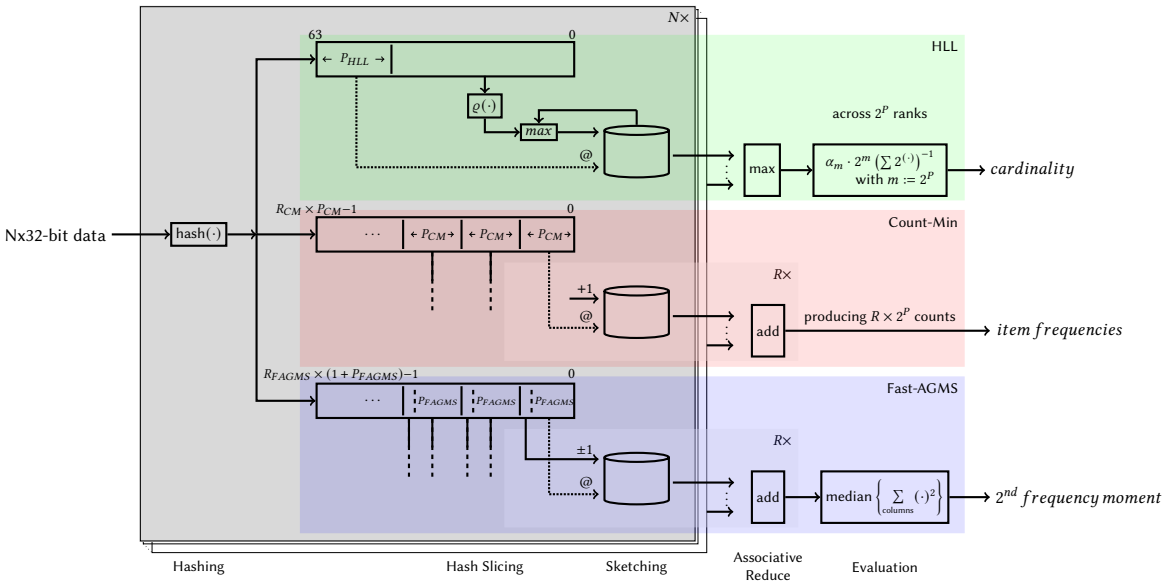


Figure 3: SKT architecture dataflow-Each pipeline consumes 32-bit of data that is hashed (Hashing); the hash value is distributed to each partial sketch and used to update the data structure behind it (Hash Slicing and Sketching). After consuming all the input values, SKT triggers the merging of all partial sketches (Associative Reduce) and generates the output results for each sketch (Evaluation).

Cormode and Garofalakis prove that this modified sketch, *Fast-AGMS*, achieves the same error bounds and confidence as the significantly more expensive original AGMS sketch. The evaluation of the sketch is only changed in that the per-row averages of squared counter values are replaced by per-row sums of squared counter values. Computationally, this even saves a division operation at the end. In SKT, we implement Fast-AGMS.

Fast-AGMS and AGMS offer the same trade-off between space and accuracy. They estimate the second frequency moment of a stream with an error of at most ε with the probabilistic confidence $(1-\delta)$ [9], where $\varepsilon = \frac{1}{2^P}$ with 2^P being the number of counters, and $\delta = \frac{1}{2^R}$ with R being the number of rows. Note the similarities with Count-Min, although we will show that the error characteristics of these two sketches are different depending on the data set analyzed.

3 SKT DESIGN

In this section, we discuss SKT's design and its implementation on an FPGA. For the exploration of the design space and as a baseline reference, we use an equally parallelizable SW-SKT targeting high-end multi-core CPUs. We consider configurations that operate (a) on data residing in the CPU memory and (b) on data arriving from the network. We also discuss the system trade-offs between sketch accuracy and the choice of the hash function and sketch sizes.

3.1 System Overview

SKT's objective is to compute the three sketches described above in a single pass over the data so as to be able to characterize a stream or data set. For this purpose, we explore a composite algorithm combining the initial hashing of the input data with three parallel sketching backends, one for each targeted metric, i.e. HLL, Count-Min and Fast-AGMS. As the update operations of these three sketches are all associative, parallel partial sketches can be computed over arbitrarily distributed sub-streams (data pipelines) before their results are merged in the output path. SKT's general architecture is shown in Figure 3. It details the architecture of one data pipeline, the merging of the computed N parallel partial sketches and the generation of the output results. The current implementation processes all the values in a data set before emitting the results computed for each sketch. While Count-Min just streams the computed sketch verbosely, HLL and Fast-AGMS perform additional computations to reduce their sketches to their final scalar estimates.

The slicing of a well randomized hash by each of the sketches can be arbitrary. We opted for an approach that results in a simple homogeneous sequence of masking extractions and constant right shifts for the software implementations of the matrix sketches. HLL is implemented according to its customary formulation relying on the *count-leading-zeros* operation after slicing off the index. This can leverage an x86 instruction in the software implementation.

The degree of parallelism and the size of the data structures is controlled by the parameters listed in Table 2. The counter matrices computed by Count-Min and Fast-AGMS are R_* rows times 2^{P_*} columns. HLL uses a linear array with $2^{P_{HLL}}$ buckets. The bit

Table 2: Design Parameters of the SKT Kernel

N	Input parallelism: threads / data pipelines (partial sketches)
$P_{\text{HLL\|CM\|FAGMS}}$	Precision: hash bits used for indexing buckets (HLL) or columns (Count-Min, Fast-AGMS)
$R_{\text{CM\|FAGMS}}$	Rows: parallel rows in the matrix sketches
$W_{\text{CM\|FAGMS}}$	Bit Width of Count-Min and Fast-AGMS counters

width of the counters is universally chosen to be $W_* = 32$. Compared to a smaller size of $W_* = 16$, this enables all sketches to cope with heavy hitters even in large data streams of a few billion items without an overflow.

The choices for the other key parameters are explored below. Larger sketches can generally be expected to produce more accurate estimates. However, they also incur higher costs both in terms of sketch storage capacity and in terms of hash computation effort.

3.2 Hash Function and Hash Size

Sketch algorithms typically seek to isolate themselves from the concrete encoding of the input by means of a randomizing hash. After hashing, they can then assume that each unique data item is assigned a fixed but seemingly random encoding drawn from the range of the chosen hash function. The statistic input randomization is the main purpose of hashing and dictates what hash function should be used.

Randomization is a stricter requirement than hash uniformity. The latter is commonly used as a hash fitness metric for implementations of hashing data containers. For example, inexpensive, and hence popular, H_3 hashes can be perfectly uniform. They map an n-bit key a from the domain $A = \mathbb{Z}_2^n$ to an m-bit hash value b of the range $B = \mathbb{Z}_2^m$ by:

$$h : A \to B$$
$$a \mapsto M \cdot a$$

M is the defining random $m \times n$-matrix over $\mathbb{Z}_2 = GF(2)$. The computation uses AND $(\cdot)$ as the multiplicative and XOR $(\oplus)$ as the additive boolean operators. As soon as the number of needed random hash bits m exceeds the number of key bits n, linear dependencies between the matrix rows become unavoidable. When that happens, hash bits become mutually dependent and patterns among input encodings reemerge in the computed hashes. The critical randomization assumption is violated.

Richter et al. [40] concluded that Murmur hashing delivers the best trade-off between performance and robustness among four hash functions (multiply-shift, multiply-add-shift, tabulation, and Murmur hashing). Kaan et al. [29] evaluate hash functions and show that simple tabulation hashing is 6.6× faster on FPGA than in software, and Murmur hashing is 1.7× faster, respectively. They also show how expensive hashing can be used in data partitioning on FPGAs without loss of performance [30]. Murmur hashing has gained a lot of popularity, with Murmur3 being widely used in practice, e.g., in Google's BigTable [25] and in research [21].

SKT uses Murmur3 as the hash. We use a software SKT implementation to validate the suitability of this choice and to determine acceptable lower bounds for the sketch sizes (Section 3.4). The number of required hash bits is determined by the maximum across all sketches implemented in SKT. Allocating no less than 64 bits to HLL [25], which leaves $64 - P_{HLL}$ for the leading-zero detection, we derive a formula that determines the number of bits to be used for hashing for all sketches:

$$H = \max\{64, R_{\text{CM}} \times P_{\text{CM}}, R_{\text{FAGMS}} \times (1 + P_{\text{FAGMS}})\} \quad (2)$$

3.3 Implementation Targets

Besides the SKT design for FPGA acceleration, we have implemented SW-SKT targeting multi-core CPUs. Both versions are implemented in C++ (conventional C++ for SW-SKT on the CPU, and High Level Synthesis [31] – HLS C++ for SKT on the FPGA). The code bases are distinct so as to optimize for each target platform. Parallelism and data reuse are thoroughly exploited in both cases. Each algorithm is parallelized over cores and threads (SW-SKT) or unfolded spatially (SKT). The hashes of incoming data items are re-used to drive the update of all three algorithms.

We use SW-SKT as a baseline, for validating the choice of hash function and hash size, and for exploring the trade-off between accuracy and sketch size. For the sake of a sound design evaluation and platform comparison, we first determine the minimum, and hence most effective, sketch size that supports one billion stream cardinality. Note that SW-SKT is a contribution on its own as it can be used in conventional systems without hardware accelerators for achieving performance gains over running each sketch separately.

3.4 The Accuracy vs. Size Trade-off

The sketch sizes are determined by the parameters R_* and P_*. In all cases (HLL, Count-Min and Fast-AGMS), the precision P_* determines the number of sketch columns (or buckets). An increase in the value of P_* reduces the chance of hash collisions and, hence, improves accuracy but also requires more space. Count-Min and Fast-AGMS take extra measures to mitigate individual hash collisions by maintaining several parallel rows with *independent* hash functions. Mutual collisions in one row are unlikely to reproduce in another. As a result, increasing R_* improves the accuracy of the overall sketch at the price of requiring more space, more memory accesses, and the computation of more hash bits. Since the final goal is to have the three sketches deployed together in a spatial architecture, it is important to understand the implications of the choices for P_* and R_* for the sketch accuracies.

To evaluate the accuracy vs. performance trade-off, we compute sketches of various dimensions over data sets with known item frequencies and pre-computed frequency moments. We create two classes of data sets of 32-bit integer elements:

- **CLS1**: This class comprises 9 data sets with uniform data distributions. Each data set is defined by a frequency $f \in \{1, 20\}$ where f is the number of times every item occurs. The maximum cardinality in this class is 1 billion and the maximum stream length is 20 billions (20 B).
- **CLS2**: This class comprises 21 data sets with Zipfian distributions with a skew $s \in \{1.5, 2.0, 3.0\}$. The maximum cardinality of this class is 1.4 millions and the maximum stream length is 1 billion (1 B).

Realistic data is frequently highly skewed [12, 36]. Intuitively, in highly skewed data, a relatively small number of distinct items appear very frequently; whereas in low-skew data, item occurrences are more uniformly distributed. Relating skew and cardinality, observe that a high skew implies a smaller cardinality in relation to a stream's length and that a high cardinality on the order of the stream length implies a low skew. The converse statements, however, do not necessarily hold true.

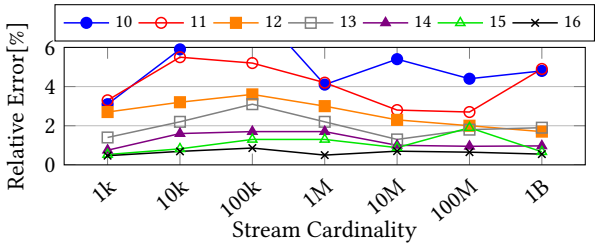


Figure 4: HLL-Relative error vs. Stream cardinality for different values of P_{HLL}. Maximum stream length is 20 B.

We study the impact on the sketch accuracy of: (1) the sketch size; (2) the input stream length; (3) the input stream cardinality. The accuracy is reported in terms of the observed relative error. We report the *maximum* relative error encountered across our class data sets, unless otherwise specified.

HLL. Figure 4 shows the relative error of HLL as a function of the stream cardinality: HLL is a very robust algorithm scaling well to large input cardinalities and sizes. A choice of $P_{HLL} \geq 14$ results in an error band of 1-2% across all input stream cardinalities. This value is aligned with other implementations and available systems as well as results reported in the literature.

Fast-AGMS. We evaluate the relative error in Fast-AGMS in different configurations. One fixes P_{FAGMS} to 13 and varies the cardinality and R_{FAGMS} (Figure 5a) and another fixes R_{FAGMS} and varies the cardinality and P_{FAGMS} (Figure 5b). Fast-AGMS is very robust to input length and cardinality scaling. All observed estimates remain within a 2% error band for $R_{FAGMS} \geq 6$ and $P_{FAGMS} \geq 13$. Nonetheless, the results show that accuracy improves by increasing row count or precision bits.

For both HLL and Fast-AGMS, the individual upper and lower peaks observed at different cardinalities (e.g., 1 M or 10 M) are an artifact of the interaction between the data and the concrete choice of hash. Changing the seed of the chosen Murmur-3 hash function displaces the jitter observed in the error graphs.

Count-Min. Unlike HLL and Fast-AGMS, Count-Min does not produce a single scalar estimate. Rather, the sketch is queried for individual item frequencies. We track the maximum query error to characterize query quality as well as the average error for a comprehensive accuracy assessment of Count-Min. Since Count-Min can be queried for items that have not appeared in the input stream, we report: (1) the relative error exclusively for items that appear in the input (Figure 6), and (2) the absolute error for all members of the 32-bit input domain (Figure 7).

Figure 6 shows that the maximum relative error increases rapidly for cardinalities larger than 1,500, whereas the average relative error increases at a slower and steadier pace. This demonstrates that Count-Min can fail *individual* queries badly. As collisions accumulate, there will eventually be an item that is affected across all rows. The overestimation even of a single low-frequency item quickly pushes the observed *maximum* relative error beyond any sensible scale. However, the *average*, and hence expected, error of sketch queries remains within acceptable bounds for larger stream cardinalities. Increasing the number of rows R_{CM} or precision bits P_{CM}, indeed, makes the sketch suitable for larger stream cardinalities, but

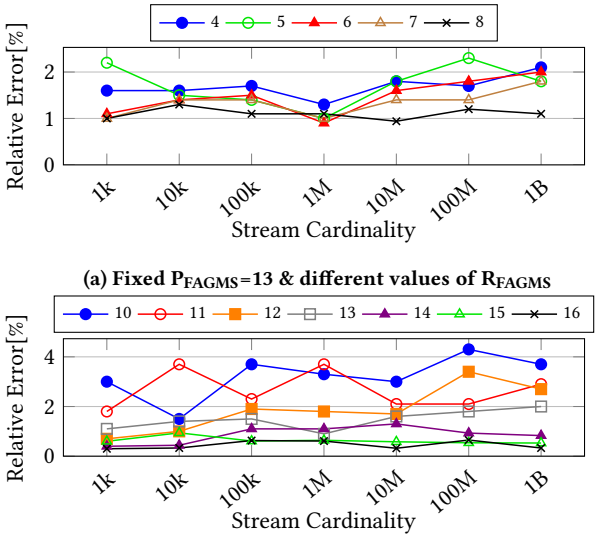


Figure 5: Fast-AGMS-Relative error vs. Stream cardinality for maximum stream length of 20 B.

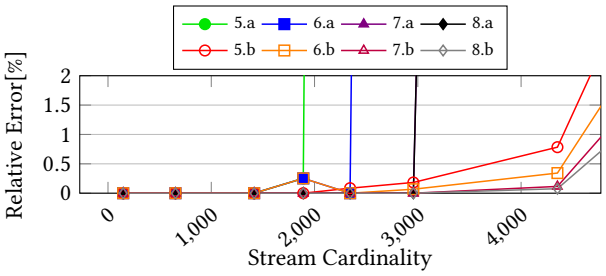


Figure 6: Count-Min-Relative error vs. Stream cardinality for fixed P_{CM}=13 & different values of R_{CM}. R_{CM}.a lines represent the maximum relative error, whereas R_{CM}.b lines represent the average of all relative errors with respect to the number of queried items.

it still remains fragile. Even computing 128 hash bits to maintain a sketch of $R_{CM} \times P_{CM} = 8 \times 16$, this is 2 MiBytes, has an average relative query error of 14.22% for a cardinality of 100,000.

A common metric to evaluate Count-Min is the average *absolute* query error as shown by Figure 7. Under this metric, the sketch appears to scale significantly better to larger input cardinalities. It is important to understand that the averaging hides that error contributions stem from rather few but significant overestimations. This underlines, once more, the accuracy compromise inherent to Count-Min. It tolerates individual significant estimation errors, which appear particularly huge on a relative scale, for an otherwise concise item frequency representation. It must be decided at application level whether or not this compromise is acceptable.

All of the quality metrics discussed indicate that the quality of Count-Min degrades as the cardinality increases. This can be explained by the growing number of distinct data items and, hence,

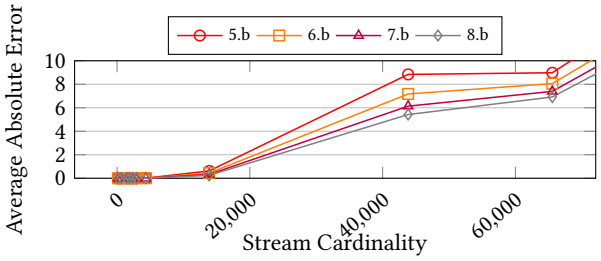


Figure 7: Count-Min-Average absolute error vs. Stream cardinality for fixed P_{CM}=13 & different values of R_{CM}. The average is computed with respect to the number of queried items.

```
1  // Instantiate Custom Key Space and Storage Type
2  static Collect<ap_uint<P_HLL>, T_RANK> collector[N];
3  #pragma HLS array_partition variable=collector dim=1

5  // Unrolled, i.e. Parallel, Operation
6  for(int i = 0; i < N; i++) {
7  #pragma HLS unroll
8      collector[i].collect(ranked[i], sketch[i],
9          // Lambda-based Update Customization
10         [](T_RANK a, T_RANK b) {
11             return std::max(a, b);
12         }
13     );
14 }
```

Figure 8: HLL Usage Example of the Collect Class

column index signatures across the sketch rows, which eventually encounter mutual overlaps in all rows. Observe that stream length alone does not pose a challenge. The fewer index signatures in a long stream with a low cardinality are more likely to avoid a collision in, at least, one of the rows. This is in line with results reported in the literature, which show that high data skew (i.e., reduced cardinality) is beneficial for the accuracy of Count-Min [12].

3.5 Practical Implications

Accuracy requirements depend on the application and the required minimum size of each sketch might differ accordingly. For our specific goal — the characterization of data streams of several billion items — we choose the sketch sizes as follows. We set $P_{\text{HLL}} = 16$ aiming at cardinality (HLL) estimation errors below 0.5%. We set $R_{\text{FAGMS}} = 6$ and $P_{\text{FAGMS}} = 13$, which yields Fast-AGMS errors below 2% for stream lengths in the order of billions. Scaling Count-Min to support large cardinalities requires too many resources on the order of this cardinality regardless of where it is implemented. Thus, we opt to match the Count-Min sketch dimension to that of Fast-AGMS. We then use HLL and Fast-AGMS to assess the accuracy of the results from Count-Min frequency. The intuition is that for large cardinalities and more uniform distributions, Count-Min will produce more errors and be less useful. The results of HLL and Fast-AGSM allow us to identify such situations and discard the results of Count-Min or, at least, indicate the potential for large individual query errors.

4 SKT FPGA IMPLEMENTATION

SKT is implemented as a customizable Vitis[1] HLS kernel [13] using a streaming architecture operating at an *initiation interval* of one. This means that the design consumes and processes inputs with every single clock cycle. SKT is completely implemented in C++. Functions and classes are templated in terms of I/O types and behavioral functors to facilitate code reuse among structurally similar modules. The kernel is customized by the parameters identified by Table 2. The parameter N is decisive for tuning the design throughput as it defines the number of inputs that are consumed in parallel in each clock cycle by the N structurally unfolded processing pipelines, each consuming one input per cycle.

[1]Xilinx' High-Level Synthesis Unified Platform Software

Note that the underlying high-level synthesis (HLS) technology is itself a compromise. It enables the generation of hardware accelerators from algorithmic descriptions on the abstraction level of systems software. This results in improved productivity over designs using register transfer level (RTL) languages. Unlike the C++ code for the CPU which abstracts away the processor architecture, the HLS code written in C++ still reflects the spatial architecture of the FPGA through the pragma directives that need to be used. These pragma directives ensure that HLS compilers produce efficient parallel designs. Figure 8 illustrates the use of pragmas to spatially unroll a loop.

4.1 Overall Design

SKT's architecture (Figure 3) implements a unidirectional data flow across a number of modules that are connected via FIFO queues, modeled as `hls::stream` objects. Throughout the design, the data is augmented with a `last` flag to mark the end of a particular stream and, thus, a job. The last flag traverses the processing pipeline together with the data both through *Hashing* and *Hash Slicing*. Its arrival at the *Sketching* memories triggers the switch from consuming sketch updates to outputting their accumulated content and resetting their state. On the output path, the N parallel, partial sketch computations are merged by *Associative Reduction*. The resulting complete sketches are further *processed* in *Evaluation* so as to compute the final results for HLL and Fast-AGMS. Finally, the results of all sketches are concatenated into a single output stream that is written to a data structure in the memory of the host CPU. The sketch results are complemented by a few trivial metrics maintained during the processing: minimum, maximum, sum, sum of squares, and count of all processed inputs.

The design consumes a customizable number of N parallel inputs in a single clock cycle. This requires that every sketch memory is able to process N updates. However, memory ports are an expensive and scarce resource. The on-chip block RAMs (BRAM) have two such ports. Considering that each update comprises both a read and a write operation, a single BRAM can only sustain the rate of a single update per clock cycle. Consequently, the parallel pipelines must construct partial sketches in *independent* memories that are merged for the final sketch output. For the same reason, SKT ensures to update the rows of the matrix sketches, Count-Min and Fast-AGMS, in parallel and assigns to each row its own independent memory.

Recall that in Figure 3 for each pipeline the forking of the input path to all the partial parallel sketches occurs after hashing the input. In software, this re-use of computed intermediates has a runtime benefit. In SKT, on the FPGA, parallel identical computations do not improve accelerator performance but increase its resource usage and power consumption footprint.

4.2 Hashing & Hash Slicing

Strong hashes, such as Murmur3, involve a series of structurally complex operations like multiplications. In RTL, the design of a such hash function would have required manual algorithm decomposition and pipelining in order to meet the throughput goals. By using Vitis HLS tool, the purely functional description of the hash computation is mapped and pipelined *automatically* to meet the targeted clock frequency and throughput goals. As the hashing is an acyclic, stateless computation, there are no systematic limits to the pipelining granularity.

After distributing the same wide hash to the individual sketches, they slice it up in different ways to derive appropriate state updates. All elementary state updates are ultimately represented as key-value pairs comprising the address of the memory location to update and an update value. In the case of HLL, the address is simply a prefix slice of the hash and the update value is its ranked tail. The update function applied to the sketch memory is the maximum with its current content. The matrix sketches, Count-Min and Fast-AGMS, feed non-overlapping parallel hash slices to the individual sketch rows. For Count-Min, a single row slice directly identifies the sketch memory location to increment. In the case of Fast-AGMS, the hash slice provides both a counter address and an extra bit determining the sign of the update step.

4.3 Sketching

The maintenance of the sketch memories is the most involved part of the design as it must explicitly account for the structural constraints on the FPGA to maximize performance. In addition to the memory port limitation, there is a state-carried data dependency that conflicts with the memory update latency of two cycles. The Vitis HLS tool needs manual assistance to circumvent these dependencies that would otherwise result in a higher initiation interval. The sketch memories, therefore, maintain a history of the most recently issued sketch updates in a shift register. If another update to an address within this history is encountered, it is based on the buffered write-back value rather than a stale read from memory. With this bypass in place, read-after-write hazards are masked and the initiation interval of a single clock cycle can be maintained. A `dependence` pragma explicitly allows Vitis HLS to disregard the data dependency carried through the memory state. These design details are not exposed to the sketch user, they are encapsulated inside a generic `Collect` class that is re-used among all SKT sketches.

Figure 8 illustrates how the `Collect` class is used by the HLL sketch inside SKT. The specialization of this class and its behavior are achieved through template parameters. They define (a) the dimension and type of the backing memory array (line 2), and (b) the state update function, i.e. maximum operation for HLL, which is specified through a lambda expression (lines 10-12). Note the custom key space, ap_uint<P_HLL>, and storage type, T_RANK. Using the HLS integral type *ap_uint<n>*, the bit width of signals and, in this case, the address space can be controlled precisely. This code instantiates N parallel memories to serve the corresponding number of data inputs (line 2). Vitis HLS must be prevented from flattening this added outer dimension by a pragma (line 3) so that designated memory ports remain available for each instance. The operational behavior is wrapped into the class member function collect , which is invoked within an unrolled and, hence, parallel loop (line 6) across all memory instances. Each instance consumes and processes updates from an hls :: stream ranked[i] until encountering an update carrying an asserted last flag. It then streams the accumulated partial sketch to the hls :: stream sketch[i].

5 EXPERIMENTAL EVALUATION

5.1 Setup

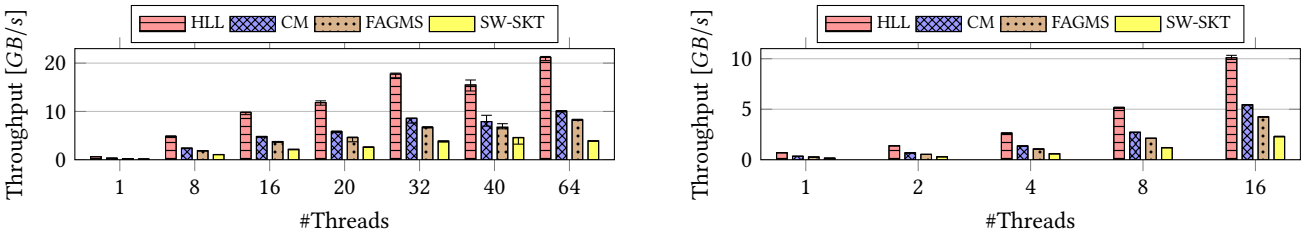
SKT has been prototyped and tested on a Xilinx Adaptive Compute Cluster (XACC) [50], a research platform for adaptive compute acceleration. SKT is evaluated on all of the cluster's heterogeneous compute resources comprising *high-end servers* (one with 2 Intel Xeon Gold 6248 with 376 GB RAM and two others with 4 Intel Xeon Gold 6234 with 376 GB RAM) as well as *Xilinx Alveo accelerator cards* (Alveo U250 and Alveo U280). We conduct RAM-to-RAM experiments for establishing performance baselines on both server platforms and for evaluating the SKT hardware acceleration. Finally, we also evaluate the direct sketching of data received over a 100 Gbps network link both in software and on the accelerator. In all experiments, the final sketch results are written back to the CPU main memory.

All measurements are conducted over data streams of 32-bit integers subjected to a randomizing 128-bit Murmur3 hash. This datatype is big enough to accommodate large cardinalities that escape their straightforward and concise characterization by histograms. It is compact enough to prevent a squashing dominance of the data IO bounds over the actual sketching performance. Larger data types make the hash computation iterate over longer input stretches and, thus, yield fewer sketch updates per input volume. A similar reductive effect occurs for extracting fields of interest from structured data. Making this field extraction runtime-programmable is trivial and would add the capability to adjust the processing to changing data schemas. While a software implementation would have to pay for this flexibility with compute time, the accelerator would be able to accommodate this functionality in a pipeline extension while impacting neither throughput nor critical sketch memory resources.

We evaluate the individual performance of each sketch algorithm (HLL, Count-Min and Fast-AGMS) using the same $P_* = 13$ in order to observe individual throughput performances. The reference dimensions for SW-SKT are $P_{\text{HLL}} = 16$ and $R_* \times P_* = 6 \times 13$ for Count-Min and Fast-AGMS.

5.2 Software Baseline

In order to assess the FPGA accelerator, it is important to understand the I/O and compute bounds of the XACC platform. First, the performance of a multi-threaded software baseline is established on the XACC cluster.



(a) Host server: Alveo0 **(b) Host server: Alveo3b**

Figure 9: CPU Baseline on two Host servers [Alveo0, Alveo3b]- Median throughput for HLL, Count-Min (CM), Fast-AGMS (FAGMS) and SW-SKT. Error bars indicate the 5th and 95th percentile. Stream length of 1 billion samples.

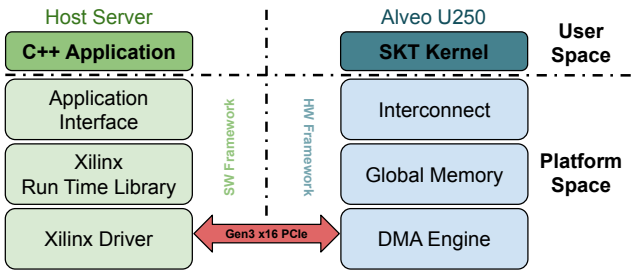


Figure 10: FPGA connected as a co-processor [Alveo U250]- QDMA Shell Platform Overview

Each of the three algorithms (HLL, Count-Min and Fast-AGMS) benefits from thread parallelism, with each thread maintaining a per-thread data structure (partial sketch) for updates. The partial sketches from all threads are only merged once all the input data has been processed. In this multi-threaded CPU implementation, threads can be equated to the parallel pipelines of the hardware design. The data flow of the CPU implementation follows the main idea behind SKT's architecture in Figure 3: 1) compute the hash value once for each data input, 2) update the corresponding partial data structures for each of the three algorithms, and 3) evaluate the sketches after processing the last stream value.

We analyze the individual performance of each of the algorithms (HLL, Count-Min and Fast-AGMS) running in isolation as well as the performance of all three algorithms running jointly (SW-SKT). The experimental platform setup of two XACC machines comprises: (*Alveo0*) 2× Intel® Xeon® Gold 6248 Processors @2.5 GHz with a combined total of 40 cores and 80 hyper-threads (Figure 9a), and (*Alveo3b*) a single Intel Xeon Gold 6234 Processor @3.3 GHz with 8 cores and 16 hyper-threads (Figure 9b). The observed sketch throughput scales with the allocation of CPU threads on each of the two machines. Figures 9a and 9b show that each algorithm's performance is correlated with the complexity of its conducted update operation. HLL performs a single update into a linear data structure. This results in the best individual performance. Count-Min and Fast-AGMS each perform $R_*\times$ more elementary counter updates than HLL. This reduces their individual performances considerably. The further difference between Count-Min and Fast-AGMS is solely due to their different update operations. While Count-Min performs *unconditional* increments on the counters identified for every row, Fast-AGMS extracts another hash bit for each counter update to *select* the sign of the increment.

When fusing all three algorithms, SW-SKT combines their compute and I/O bounds and attains a peak throughput of 4.56 GBytes/s for 40 compute threads, i.e., the physical core count of the Alveo0 machine. While all individual sketches and their combination in SW-SKT compute the same input hash, their throughput are clearly differentiated by the performed sketch updates. The observed performance decreases with the number of required counter updates and with the complexity of the increment operations to perform.

5.3 QDMA Integration

For the first evaluation on the FPGA, the SKT kernel is interfaced directly from the host as a free-running OpenCL kernel within the Xilinx QDMA shell on an Alveo U250 accelerator card operated under Vitis 2020.2. The Alveo U250 deployment environment together with the software framework running on the host server are illustrated in Figure 10. The kernel is configured to process streams comprising 32-bit data items, which are hashed by a strong 128-bit Murmur3 hash. The design operates at the default platform clock of 300 MHz. The input parallelism can be scaled up to 16 parallel data lanes fit in by the 512-bit user kernel interface. This parallelism and the sketch dimensions are explored and evaluated experimentally.

Throughput Scaling. A single data pipeline processing 32 bit inputs at 300 MHz consumes a bandwidth of 1.2 GByte/s. We aim at matching the number N of parallel pipelines with the maximum kernel interface bandwidth. The maximum theoretical bandwidth for the PCIe Gen3 ×16 connection on the Alveo U250 card is 15.62 GByte/s. Bus interference and protocol overhead imposed by both PCIe and the QDMA shell reduce the actually achieved figure. Starting the exploration with $N = 16$ parallel pipelines ensures that we identify the actual limit of the platform rather than a bottleneck in SKT implementation. Experiments are run from small stream sizes of 1000 up to streams of 500 million data items. This is just below the job size limit of 2 GiByte imposed by the QDMA shell. Currently, the shell driver truncates any larger job *silently*. A scalar kernel parameter or a custom streaming protocol layer could be added to aggregate multiple QDMA transmissions before evaluating the collected sketches. Then, larger jobs could be partitioned and handled by multiple QDMA invocations. Yet another service management layer would have to ensure the integrity and

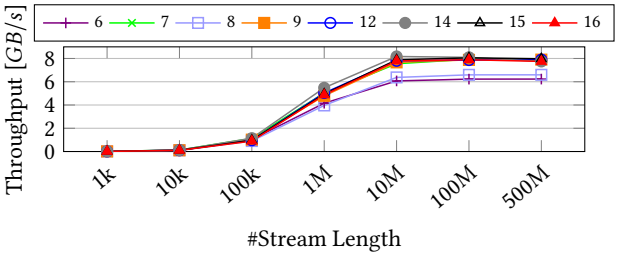


Figure 11: SKT on QDMA Shell-FPGA as a co-processor. Mean throughput observed from the CPU.

atomicity of these partitioned jobs in contended use contexts. Figure 11 shows the data throughput achieved for these experiments as measured in the host application from initiating the streaming to the accelerator to completing the reception of the concatenated computed sketches. Hence, this measurement includes the communication initiation overhead as well as the sketch evaluation and transmission times, which are all independent of the size of the input stream. Actual observations experience a variation of ±5% around the shown means.

Observe that the Count-Min sketch, which is transmitted verbosely, already has a size of $R_{CM} \times 2^{P_{CM}}$ 32-bit words. Its write back completely dominates the observed accelerator latency for inputs up to 10^5 items. For larger stream sizes, the impact such one-time costs diminishes. The throughput is observed to saturate around 8 GByte/s even for $N = 16$. In conclusion, only a reduction down to $N = 6$ parallel processing pipelines would be expected to establish a processing bottleneck. However, far more subtle reductions of parallelism were causing notable performance degradations until we were investing on-chip longer-latency UltraRAM resources to implement elastic FIFOs at the interface between the QDMA shell and the SKT kernel. Apparently, the assertion of backpressure across this boundary causes disadvantageous interactions with the flow control of the QDMA infrastructure. While the FIFOs eliminated this effect completely for most settings, a curious performance dip remained reproducible for $N = 8$ parallel processing pipelines. The degradation of the performance for $N = 6$ is, finally, expected. The throughput of 8 GByte/s that is achieved by 9 or more pipelines is 1.75× higher than what the 40 cores of the dual-processor *Alveo0* system were able to attain.

Sketch Optimization. Table 3 shows the utilization of the FPGA hardware resources, which are available to the user kernel, by selected SKT configurations. Note that SKT only consumes a small fraction of the general-purpose combinatorial and sequential fabric logic, i.e. lookup tables (LUTs) and registers (Regs), respectively.

Table 3: User Budget Resource Utilization on Alveo U250

$N \times R_* \times P_*$	LUT	Reg	BRAM	λ	DSP
$6 \times 6 \times 13$	43082 (2.8%)	63279 (2.0%)	624 (25.1%)	0.93	476 (3.9%)
$9 \times 6 \times 13$	59614 (3.9%)	86216 (2.7%)	936 (37.7%)	0.93	662 (5.4%)
$16 \times 6 \times 13$	91530 (6.0%)	138945 (4.4%)	1664 (67.0%)	0.93	1096 (8.9%)
$16 \times 8 \times 13$	115851 (7.7%)	172030 (5.4%)	2144 (86.4%)	0.93	1104 (9.0%)

Also, the utilization of arithmetic DSP slices is moderate (<10%), with less than 1% being consumed for the sketch evaluations, and the rest being used by the hash computation. Hence, there is the stark correlation with the number N of input lanes. The critical and limiting resources type is the on-chip memory provided by Block RAM (BRAM) tiles. As shown in Table 3, the QDMA integration of SKT can scale beyond the fixed sketch size of $P_{\text{HLL}} = 16$ and $R_* \times P_* = 6 \times 13$ even for $N = 16$ parallel pipelines.

To utilize memory resources optimally, a good understanding of their demand in relation to the kernel parameters is beneficial. Individual BRAM tiles serve a 10-bit address space with 36-bit data. Larger address spaces are assembled from multiple memory tiles. In an RTL design a traditional manual memory allocation would use one memory location for each of the 32-bit counters for Count-Min and Fast-AGMS, leading to a higher resource consumption. In HLS, the memory allocation is done automatically and can be optimized by the tools. Instead of allocating one memory location for each counter, the tools avoid memory fragmentation, claiming storage space more optimally. This is seen especially for the matrix sketches whose 32-bit counters are dominating the memory resource consumption figure. For the HLL sketch, the memory geometry of each BRAM tile is reshaped into a 12-bit address space of 9-bit data. This suffices to accommodate the ranks whose range is bounded by the bit width of the consumed hash value. This layout transformation is directly supported by the physical BRAM structures. All these considerations give rise to the BRAM utilization model in Equation 3 for sketch parameter settings with $P_{\text{HLL}} \geq 12$ and $P_{\text{CM}}, P_{\text{AGMS}} \geq 10$:

$$U_{\text{BRAM}} = \lambda \cdot N \cdot \left(R_{AGMS} \cdot 2^{P_{AGMS}-10} + R_{CM} \cdot 2^{P_{CM}-10} + 2^{P_{HLL}-12} \right) \quad (3)$$
$$\text{with} \quad \frac{W}{36} \leq \lambda \leq 1$$

The model introduces an extra coefficient λ that reflects the automated memory layout optimization of the HLS. An optimal recycling of all 32-bit counters memory fragmentation would, at best, allow the reduction of the memory footprint to $\lambda_{\text{min}} = \frac{32}{36} \sim 0.89$.

The λ(s) are tabulated with the designs in Table 3. They demonstrate that Vitis HLS is able to assemble BRAM resources for the sketch storage very efficiently, close to the projected optimum. However, it must also be noted that Vitis was unable to complete the hardware implementation for any sketch design with a projected BRAM tile utilization above 90%. This blockade is due to signal routing challenges inside the FPGA that put the tools into a boundless optimisation problem. A way out is to resort to the optimization techniques available in RTL but this would defeat the purpose of implementing the system as a high-level design.

In summary, a SKT implementation that serves the full 512-bit kernel interface with sketch dimensions of $P_{\text{HLL}} = 16$ and $R_* \times P_* = 6 \times 13$ is clearly feasible. There is room to increase the row count to $R_* = 8$ directly. Even larger sketches are attainable by reducing the number of parallel pipelines N. There are no observable throughput penalties as long as $N \geq 9$. The developed utilization model for the critical BRAM resources is valid across all modern Xilinx platforms and also guides the assessment of the network integration.

5.4 Network Integration

For the TCP/IP network experiment, SKT is used as a free-running OpenCL kernel within the Xilinx XDMA shell on the Alveo U280

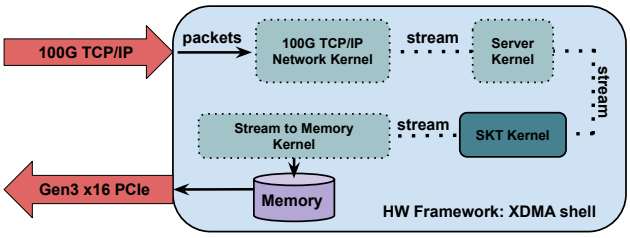


Figure 12: FPGA connected on the TCP/IP network [Alveo U280]-XDMA Shell instantiation of User Space Kernels

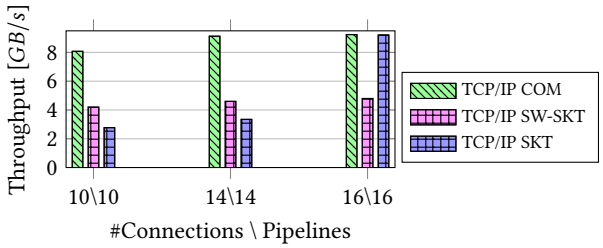


Figure 13: Mean TCP/IP throughput for remote communication (COM) and remote sketching over 1 billion samples by SW-SKT (Alveo0) and by SKT inside XDMA shell (Alveo U280 FPGA connected to Alveo3b)

accelerator card. The shell and accelerator card differ from the ones used for SKT's stand-alone evaluation above. While the ability to stream data from the CPU is currently only available in the QDMA shell on Alveo U250, the 100G TCP/IP network stack [42] is currently only available under the XDMA shell on the Alveo U280 platform [22, 23]. Hence, for these experiments, we use the XDMA shell. Feeding the data from the network avoids the job size limit imposed by the QDMA shell. As the two shells run on different boards, U250 for QDMA and U280 for XDMA, SKT faces different operating conditions. First, the kernel clock frequency of 250 MHz is lower within the XDMA shell. Second, there are fewer available BRAM tiles. In order to fit onto the U280, SKT needs to shrink. To implement the 16 parallel pipelines needed to sustain the 100 Gbps line rate, we need to reduce the number of rows for the matrix sketches to $R_* = 5$. We do so for the purposes of the experimental evaluation here. This limitation will disappear soon as new versions of the shells become available to support the whole spectrum of boards alike. For the sake of completeness of the experiments, we include these results here.

Figure 12 illustrates the chain of kernels used by our system. It processes the data received from the 100G TCP/IP network and passes the results to the server through a Gen3 ×16 PCIe connection. The *100G TCP/IP Network Kernel* instantiates the 100G Ethernet subsystem, provides TCP/IP functionality, and converts the data packets to a flat data stream. The *Server Kernel* listens on the network, accepts TCP/IP connections, and forwards the incoming data to the SKT kernel. When all data has arrived, the server kernel asserts the last signal communicating the completion of a job so as to trigger the result generation. The *SKT Kernel* consumes the received data stream and generates a sketch stream upon job completion. The output is sent to the host CPU memory by the *Stream to Memory Kernel*, from where the results are read by the host server via the OpenCL API. This last kernel is needed as an adapter to the XDMA shell, which does not support streaming natively. Except for the free-running SKT kernel, all the other three kernels are explicitly controlled by the application running on the host server.

Our network experiments are summarized in Figure 13. They show that SKT needs 16 pipelines in order to support the 100 Gbps network line-rate. Whenever fewer pipelines are allocated, e.g., 14 or 10, the sustained line-rates drop sharply to 3.35 GB/s and 2.77 GB/s, respectively. This drop is caused by the way the 100G TCP/IP network kernel expects its output to be consumed. More specifically, the network kernel expects the following kernel to consume 512-bit data, irrespective of the line-rate. This amounts to 16 parallel lanes of 32-bit data. If SKT, for example, only implements 14 parallel pipelines, it consumes 448-bit in every cycle, postponing the remaining 64 bits in a buffer. At 100 Gbps, the accumulation of the buffered data eventually triggers the flow control of the network kernel. This mechanism relies on external memory whose throughput within the XDMA shell is much lower than 100 Gbps. We expect this limitation to disappear soon in future systems.

For comparison, SW-SKT is used to sketch the same data received over the network on the 40-core Alveo0 machine. So as to eliminate the TCP/IP stack of the OS as a bottleneck, the traffic is split across parallel TCP/IP connections. A designated receiver thread is forked to service each of these connections. Without any processing of the received data, this setup is able to sustain a data throughput of almost 10 GB/s as shown by the columns labeled "COM" in Figure 13. The actual sketching is challenging enough to warrant another level of workload distribution. We show the highest throughput figures, which are obtained by backing each receiver thread by four compute threads. The receiver only delegates data blocks through a job queue and guarantees a steady consumption of data from the TCP/IP socket. As in the FPGA-accelerated system, a throttling of the data input by backpressure proved to impact the achievable system performance negatively. An overprovisioning of TCP/IP connections was not able to mitigate this effect. In the background, each compute thread maintains its own partial sketch so as to confine the synchronization overhead to the conclusive sketch merger. As shown by the SW-SKT in Figure 9, even the parallelization of the sketch computation across all cores fall short of attaining a processing throughput of 5 GB/s. This is consistent with the performance observed for the RAM-to-RAM deployment of SW-SKT but only half of the performance achieved by the network-facing FPGA accelerator.

6 DISCUSSIONS

6.1 HLS Experience

Finally, we evaluate our experience from using high-level synthesis with a C++ hardware design entry. This approach comes with clear productivity benefits, in particular:

- An easier *accessibility* for software engineers.
- A ready application *integration* stack.

- Automated *pipelining* for high accelerator performance.
- Automated *memory layout optimization* beyond what would be reasonably maintainable in a manual RTL design.

The price paid for these gains is best illustrated by the device resource utilization figures achieved. Sketch sizes that would demand more than 90% of the user budget of BRAM tiles failed consistently to implement. Mutually amplifying utilization and timing challenges are not uncommon in classic hardware design in RTL. However, the added layers of abstraction in HLS severely limit the ability to resolve such situations. Engaging in floorplanning (where each part of the design goes), manual placement, or even manual signal routing would clearly defeat the purpose of using HLS.

6.2 Related Work

The technological evolution of FPGAs has enabled the integration of more and more resources on FPGA devices. This has opened the door for deploying increasingly complex designs, such as SKT, on FPGA accelerators. SKT is a complete data center solution that has been evaluated in an integrated end-to-end system context. There are several other recent works that analyze and evaluate the performance of sketch algorithms on FPGAs.

Kulkarni et al. [33] implement HLL and use it to process streams received through a 100 Gbps network. They use the entire FPGA for the HLL design. In contrast, SKT adjusts the design of the HyperLogLog sketch computation to still match the network bandwidth but leaving space for the computation of other sketches *in parallel*. Kulkarni et al. also demonstrate empirically the need to use a 64-bit Murmur3 hash to improve the cardinality estimation accuracy for larger data streams. In this paper, we expand the range of configurations considered and provide a far more detailed analysis of the throughput-accuracy-resources trade-offs.

Scotch [32] is a VHDL-based code generator for sketch implementations. Like SKT, Scotch [32] also aims at maximizing the use of the available on-chip memory. However, it does so to implement a *single* custom sketch out of the supported sketch classes. While the plain scaling of a single sketch faces diminishing returns, the multi-sketch approach pioneered by SKT uses the resources to deliver extra value. Scotch and SKT also differ in the chosen design methodology. Scotch relies on traditional RTL design. Hand-coded VHDL code is patched by a generator tool to match a custom sketch specification. Scotch explores the largest feasible sketch dimension only during the design implementation, putting the complete synthesis toolchain into the exploration loop. SKT avoids this huge, often multi-day effort by providing a utilization model for the critical BRAM resources. Last but not least, Scotch forgoes an end-to-end in-system evaluation. Its performance results do not include data transmission and result extraction overheads although the integration with the host application pose decisive practical bounds to the performance that can be extracted from a hardware accelerator. A further important difference in terms of design and accuracy is that Scotch uses H_3 hashes while SKT uses Murmur3 hashes.

Tong and Prasanna have proposed an FPGA-based, RTL-designed sketch accelerator [46] for monitoring and detecting network traffic anomalies [47]. Theirs is a Count-Min sketch, over which they support Count-Min and K-ary queries. In contrast to SKT, which particularly accelerates the sketch constructions, Tong and Prasanna monitor individual sketch updates to identify anomalies, i.e., heavy hitters (Count-Min) or heavy change (K-ary), on the fly. They never communicate the total accumulated sketch itself. Like Scotch, their implementation is also based on H_3 hashes.

In the same way that we use SKT for characterizing streams, sketches can also be used for the structural analysis of multidimensional data. Rouhani et al. [41] describe SSketch as a framework for building streaming analysis accelerators on FPGAs for this purpose. The design works on a 1 Gbps network while SKT targets 100 Gbps, a throughput higher by two orders of magnitude.

6.3 Availability

SKT is open source. Both the implementation of the accelerator kernel and of the software reference are available on GitHub: https://github.com/fpgasystems/skt.

7 CONCLUSIONS

Characterizing data sets and streams has become a fundamental operation in a wide range of applications. Developing a better understanding of the data at hand involves obtaining information about aspects such as the cardinality or the distribution of the data set. In the paper, we show that this can be an expensive operation on conventional CPUs, even when parallelizing across many cores and processors, both when the data is in the machine or when it is streamed over the network. Thus, we propose SKT, a one-pass, multi-sketch accelerator implemented on an FPGA. SKT computes three widely used sketch algorithms, HyperLogLog, Count-Min and Fast-AGMS, that complement each other in terms of the information they provide and two of which (HyperLogLog and Fast-AGMS) can be used to identify data distributions where the accuracy of Count-Min is compromised. The architecture of SKT is based on an extensive design space exploration to identify the interesting trade-offs in terms of performance, resource utilization, and accuracy. The experimental evaluation demonstrates that SKT running on a single FPGA can match the performance of 70 CPU cores. Moreover, SKT also been deployed into a smart NIC to show its ability to process data streams at 100 Gbps directly from the network. Finally, the use of FPGAs is always a question mark due to the complexity added over software programming. SKT has been programmed entirely in High Level Synthesis (HLS), using a version of C++ that is accessible to software programmers as many of the details of the FPGA architecture are hidden behind pragmas and high-level abstractions provided by the development environment. In the paper, we provide detailed explanations on when and how to make HLS behave as needed to match the needs of the design. In addition to an effective solution to the data characterization problem, we thus also provide general guidelines for designers interested in taking advantage of the heterogeneous, special-purpose hardware acceleration that is defining today's advances in massive data processing.

ACKNOWLEDGMENTS

We thank Xilinx for their generous donation of the Xilinx Adaptive Compute Cluster (XACC) equipped with their recent Alveo U250 and Alveo U280 accelerator technology. We also thank Zhenhao He of ETH Zürich for his valuable support in integrating the Xilinx XDMA shell with the TCP/IP network protocol stack.

REFERENCES

[1] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. 1999. Tracking Join and Self-Join Sizes in Limited Storage. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS '99)*. 10–20. https://doi.org/10.1145/303976.303978

[2] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC '96)*. 20–29. https://doi.org/10.1145/237814.237823

[3] AWSCloud. 2020. *AQUA (Advanced Query Accelerator) for Amazon Redshift*. Retrieved December 21, 2020 from https://pages.awscloud.com/AQUA_Preview.html

[4] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. 2012. Four degrees of separation. In *Proceedings of the 4th Annual ACM Web Science Conference (WebSci '12)*. 33–42. https://doi.org/10.1145/2380718.2380723

[5] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[6] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 511–519.

[7] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20. https://doi.org/10.1109/MM.2018.022071131

[8] Graham Cormode. 2017. Data Sketching. *Commun. ACM* 60, 9 (2017), 48–55. https://doi.org/10.1145/3080008

[9] Graham Cormode and Minos Garofalakis. 2005. Sketching Streams through the Net: Distributed Approximate Query Tracking. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. 13–24.

[10] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. 2012. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases* 4, 1-3 (2012), 1–294. https://doi.org/10.1561/1900000004

[11] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the Count-Min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[12] Graham Cormode and Shan Muthukrishnan. 2005. Summarizing and mining skewed data streams. In *Proceedings of the SIAM International Conference on Data Mining (SDM '05)*. 44–55.

[13] L.H. Crockett, S.A. Elliot, M.A. Enderwitz, and R.W. Stewart. 2014. *The Zynq Book* (1. ed.). Strathclyde Academic Media, UK.

[14] Bill Dally. 2011. Power, programmability, and granularity: The challenges of exascale computing. In *IEEE International Test Conference*. IEEE Computer Society, 12–12.

[15] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (2020), 48–57. https://doi.org/10.1145/3361682

[16] Apache DataSketches. 2021. *The Business Challenge: Analyzing Big Data Quickly*. Retrieved January 20, 2021 from https://datasketches.apache.org/docs/Background/TheChallenge.html

[17] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 1998. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. 28 (1998), 254–265. https://doi.org/10.1145/285237.285287

[18] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*. 51–66.

[19] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm. In *Discrete Mathematics and Theoretical Computer Science (DMTCS '07)*. 137–156.

[20] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*. 1–14.

[21] Marc Antoine Gosselin-Lavigne, Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. 2015. A performance evaluation of hash functions for IP reputation lookup using bloom filters. In *10th International Conference on Availability, Reliability and Security (ARES '15)*. 516–521. https://doi.org/10.1109/ARES.2015.101

[22] Zhenhao He. 2020. *Vitis with 100 Gbps TCP/IP*. Retrieved January 20, 2021 from https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP

[23] Zhenhao He, Dario Korolija, and Gustavo Alonso. 2021. EasyNet: 100 Gbps Network for HLS. In *International Conference on Field-Programmable Logic and Applications (FPL '21)*. https://doi.org/10.3929/ethz-b-000487920

[24] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (2019), 48–60. https://doi.org/10.1145/3282307

[25] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. 683–692. https://doi.org/10.1145/2452376.2452456

[26] Felipe Hoffa. 2017. *Counting uniques faster in BigQuery with HyperLogLog++*. Retrieved December 21, 2020 from https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog

[27] Akanksha Jain and Calvin Lin. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*. 78–89.

[28] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment* 9, 12 (2016), 924–935. https://doi.org/10.14778/2994509.2994512

[29] Kaan Kara and Gustavo Alonso. 2016. Fast and robust hashing for database operators. In *26th International Conference on Field Programmable Logic and Applications (FPL '16)*. 1–4.

[30] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-based data partitioning. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '17)*. 433–445. https://doi.org/10.1145/3035918.3035946

[31] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel Programming for FPGAs. *arXiv preprint arXiv:1805.03648* (2018).

[32] Martin Kiefer, Ilias Poulakis, Sebastian Breß, and Volker Markl. 2020. Scotch: Generating FPGA-Accelerators for Sketching at Line Rate. *Proceedings of the VLDB Endowment* 14, 3 (2020), 281–293.

[33] Amit Kulkarni, Monica Chiosa, Thomas B. Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. 2020. HyperLogLog Sketch Acceleration on FPGA. In *International Conference on Field Programmable Logic and Applications (FPL '20)*. 47–56. https://doi.org/10.1109/FPL50879.2020.00019

[34] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[35] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 137–152. https://doi.org/10.1145/3132747.3132756

[36] Nishad Manerikar and Themis Palpanas. 2009. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering* 68, 4 (2009), 415–430. https://doi.org/10.1016/j.datak.2008.11.001

[37] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2008. Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In *Proceedings of the 11th International Conference on Extending Database Technology (EDBT '08)*. 618–629. https://doi.org/10.1145/1353343.1353418

[38] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *IEEE 27th International Conference on Data Engineering (ICDE '11)*. 984–994.

[39] Amazon Redshift. [n.d.]. *Using HyperLogLog sketches in Amazon Redshift*. Retrieved December 21, 2020 from https://docs.aws.amazon.com/redshift/latest/dg/hyperloglog-overview.html

[40] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proceedings of the VLDB Endowment* 9, 3 (2015), 96–107.

[41] Bita D. Rouhani, Ebrahim M. Songhori, Azalia Mirhoseini, and Farinaz Koushanfar. 2015. SSketch: An Automated Framework for Streaming Sketch-Based Analysis of Big Data on FPGA. In *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '15)*. 187–194. https://doi.org/10.1109/FCCM.2015.56

[42] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *29th International Conference on Field Programmable Logic and Applications (FPL '19)*. 286–292.

[43] Florin Rusu and Alin Dobra. 2007. Statistical analysis of sketch estimators. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '07)*. 187–198. https://doi.org/10.1145/1247480.1247503

[44] Sim Simeonov. 2019. *Advanced Analytics with HyperLogLog Functions in Apache Spark*. Retrieved January 30, 2021 from https://databricks.com/blog/2019/05/08/advanced-analytics-with-apache-spark.html

[45] Yanwei Song and Engin Ipek. 2015. More is less: Improving the energy efficiency of data movement via opportunistic use of sparse codes. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '15)*. 242–254.

[46] Da Tong and Viktor K. Prasanna. 2016. High Throughput Sketch Based Online Heavy Hitter Detection on FPGA. *SIGARCH Comput. Archit. News* 43, 4 (2016), 70–75. https://doi.org/10.1145/2927964.2927977

[47] Da Tong and Viktor K. Prasanna. 2018. Sketch Acceleration on FPGA and its Applications in Network Anomaly Detection. *IEEE Transactions on Parallel and*

Distributed Systems 29, 4 (2018), 929–942. https://doi.org/10.1109/TPDS.2017.2766633

[48] Shibo Wang and Engin Ipek. 2016. Reducing data movement energy via online data clustering and encoding. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*. 1–13.

[49] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment* 7, 11 (2014), 963–974. https://doi.org/10.14778/2732967.2732972

[50] Xilinx. [n.d.]. *Xilinx Adaptive Compute Cluster (XACC) Program.* Retrieved December 21, 2020 from https://www.xilinx.com/support/university/XUP-XACC.html

[51] Xin Zhang, Chang Lan, and Adrian Perrig. 2012. Secure and Scalable Fault Localization under Dynamic Traffic Patterns. In *IEEE Symposium on Security and Privacy (IEEE '12)*. 317–331. https://doi.org/10.1109/SP.2012.27

[52] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. 2004. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement (IMC '04)*. 101–114. https://doi.org/10.1145/1028788.1028802

[53] Zhenjie Zhang, Yin Yang, Ruichu Cai, Dimitris Papadias, and Anthony Tung. 2009. Kernel-Based Skyline Cardinality Estimation. In *Proceedings of the 2009 ACM International Conference on Management of Data (SIGMOD '09)*. 509–522. https://doi.org/10.1145/1559845.1559899