In-Storage Computation of Histograms with Differential Privacy

Andrei Tosa, Anca Hangan, Gheorghe Sebestyen
Technical University of Cluj-Napoca, Romania
{firstname.lastname}@cs.utcluj.ro

Zsolt István
TU Darmstadt, Germany
zsolt.istvan@cs.tu-darmstadt.de

***Abstract*—Network-attached Smart Storage is becoming increasingly common in data analytics applications. It relies on processing elements, such as FPGAs, close to the storage medium to offload compute-intensive operations, reducing data movement across distributed nodes in the system. As a result, it can offer outstanding performance and energy efficiency.**

Modern data analytics systems are not only becoming more distributed they are also increasingly focusing on privacy policy compliance. This means that, in the future, Smart Storage will have to offload more and more privacy-related processing. In this work, we explore how the computation of differentially private (DP) histograms, a basic building block of privacy-preserving analytics, can be offloaded to FPGAs. By performing DP aggregation on the storage side, untrusted clients can be allowed to query the data in aggregate form without risking the leakage of personally identifiable information.

We prototype our idea by extending an FPGA-based distributed key-value store with three new components. First, a histogram module, that processes values at 100Gbps line-rate. Second, a random noise generator that adds noise to final histogram according to the rules dictated by DP. Third, a mechanism to limit the rate at which key-value pairs can be used in histograms, to stay within the DP privacy budget.

I. Introduction

Data analytics applications in the cloud and datacenters today run in a distributed manner, separating compute nodes from storage nodes to increase elasticity and to be able to scale the compute resources independent from storage capacity. This architecture, even though overall beneficial, suffers from various data movement bottlenecks. Networked Smart Storage, such as Amazon Aqua [1], is an emerging option to provide query offload close to the data source, reducing data movement bottlenecks and speeding up overall processing.

With the emergence of stricter privacy regulation, such as GDPR, data analytics and storage face a novel set of challenges [11]. Recent work in the space of Smart Storage shows that, beyond challenges, there are also opportunities to be had [5]. One such opportunity is in implementing some form of privacy preserving computation inside the storage. There are many types of processing that balance data utility and user privacy, with Differential Privacy (DP) [4] being one widely-used example. DP is widely used for releasing statistics, such as histograms or aggregates about a collection of persons without exposing the contribution of an individual's data in the final result. Histograms with DP are especially often used in practice, for instance, by Google in various contexts [2], [13] and the US Census Bureau.

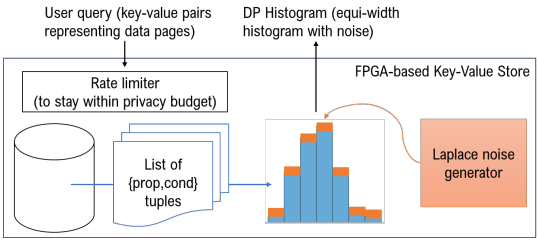

Fig. 1. To compute differentially private histograms, regular histograms are computed on the data and noise is added to the counts. The frequency at which such histograms can be created on a dataset has to be controlled, to disallow statistical attacks on the results.

Motivated by the above, we explore how one can offload differentially private histogram computation to an FPGA in a Smart Storage node. Our prototype builds on an open-source key-value store and exposes two interfaces: a traditional get/set interface to the actual data for privileged users and a separate, histogram-based interface for non-privileged users. By the nature of DP, the modules we implement will not be in use continuously – in DP, there is a limit to how often queries on the same dataset can be repeated before depleting the so called privacy budget. Nonetheless, to avoid slowdowns in other, privileged queries, we designed our modules to be able to handle the 100Gbps data rate of the key-value store. We implement our prototype as three modules (see Figure 1):

- A module that computes equi-width histograms on tuples, made up of a numerical "property" field and a "condition" field. This module uses multiple parallel pipelines to accelerate processing.
- A noise generator with Laplace distribution as required by DP. This module is composed of a pluggable random number generator and a distribution mapper.
- A mechanism to limit the frequency with which key-value pairs can be accessed by histogram queries issued by unprivileged users (i.e., rate limiting). This mechanism is necessary to ensure that DP privacy budgets are respected.

II. Background and Related Work

A. Differential Privacy and Histograms

Differential Privacy (DP) [4] allows publishing statistics about sensitive datasets in a way that hides the information about specific individuals in that dataset. The underlying idea of DP is that if each individual's contribution has only a small

effect on the final statistical results, this can be masked by adding a carefully chosen amount of noise that maximizes data utility while also protecting privacy.

In this work we focus on histograms computed on numerical data, for instance, the occurrence of a disease or condition in an age group or income level [14], or the count of people in a specific shop at different times during the day [13]. In order to make a histogram DP, noise needs to be added to each category after computation[1]. This noise is sampled from a Laplace distribution, with a mean of zero. The magnitude of the noise is a function of the ϵ sensitivity parameter chosen – plainly put, the larger the magnitude, the better the privacy protection but the lower the utility of the resulting histogram. The choice of ϵ will determine how much private information is "leaked" on each re-generation of a histogram and together with the "privacy budget" determines how often a specific dataset can be used to generate statistics. The mechanisms determining the privacy budget and the choice of ϵ are orthogonal to this paper but there is related work that provides practical examples [2].

Computation of histograms on database pages and key-value store data has been explored in various related work [6], [9], [10]. The main challenge of computing histograms on FPGAs is related to handling numerical data spread across a wide range (e.g., 64bit integers) where it's infeasible to maintain a count at very small granularity – hence some form of normalization or coarser grained view is needed. Related work showed that histogram creation can only be partially parallelized for higher throughput because a final aggregation of partial sums is always necessary [6]. In this work, we adopt best practices of histogram computation.

B. Smart Storage and Multes

Smart Storage has been the subject of research for decades [3], [7]–[9] and today, in an effort to alleviate data movement bottlenecks in Big Data analytics, Smart Storage is being actively used in the cloud. Many of these solutions rely on FPGAs and expose a key-value store interface. Notable examples are Amazon AWS Aqua [1] and Samsung SmartSSDs.

Our prototype extends Multes, an open-source FPGA-based key-value store that exposes a 100Gbps TCP/IP network interface [2]. Its high level architecture can be seen in Figure 2: it is composed of 1) a networking module (running at 100Gbps rate on a Xilinx Alveo U280 board), 2) a Cuckoo Hash Table module, that stores keys and pointer to the "value storage area", 3) the Value Store logic that reads and writes values to storage (in the case of the U280, to HBM memory), and 4) a streaming interface for plugging in user processing.

Multes exposes typical GET/SET/DELETE operations over the network, to software clients. The user-defined processing can be invoked with GETCOND operations: these resemble a GET request but can also transmit parameters to the processing modules. The user-defined module receives values read by the GETCOND command over a 512bit streaming interface and can return arbitrary transformed data to the clients.

[1]We focus on equi-width histograms whose categories can be considered public knowledge. There are more sophisticated schemes in the related work for other scenarios.

[2]https://github.com/zistvan/Multes_for_Vitis_with_100Gbps_TCP-IP

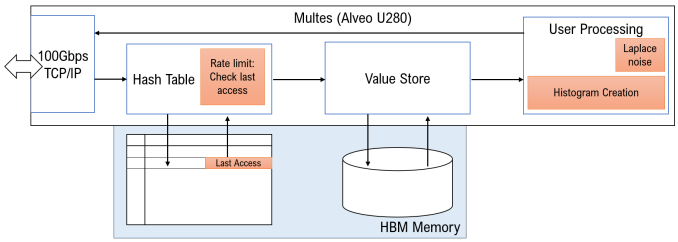

Fig. 2. Multes is a 100Gbps KVS, running on a Xilinx Alveo U280 card, using HBM memory for storing key-value pairs. We extend its functionality with three modules, highlighted in orange.

Multes has a leaky-bucket-based performance isolation functionality to provide QoS to multiple key-value store clients. This mechanism acts as a rate limiter on the number of operations performed per second (and the bandwidth usage) for each client. The rate limiting functionality required for the DP prototype is similar but it has to ensure globally that keys are not read too often by histogram-creation queries.

III. Implementation

As can be seen highlighted in Figure 2, we extend Multes with three modules: 1) equi-width histogram module, 2) Laplace noise generator and 3) rate limiting for reads used in histogram creation. We rely on the existing connection points for user processing to add a histogram module with its corresponding noise generator, and we extend the hash table to perform read rate limiting for GETCOND operations.

For the purpose of this work, we will assume that the values stored in the key-value store contain lists of $\{\text{property}, \text{category}\}$ tuples. The *property* is a 32bit numerical property of an individual (e.g., age, income, height, etc.) and *category* is a 32bit flag that describes the categories an individual belongs to (e.g., smoker or not, suffering from a specific disease, etc.). The histograms are then computed on all tuples that have a specific value for their *category*, making buckets of the ranges of *property* and counting occurrences in each bucket. To create a histogram, the lists of tuples in several key-value pairs (each in the KB range) can be merged by reading them iteratively with GETCOND. These operations can also encode the "last request" so the histogram module can generate its final output.

A. Equi-Width Histogram Computation

The computation of the histogram is performed in several parallel and pipelined steps, as shown in Figure 3. Value data is received from the Multes pipeline on a 512bit streaming interface which is split in 64bit tuples, fed into several data-parallel pipelines. Tuples consist of a 32bit property and a 32bit flag part. The histogram module is parameterized at the beginning of the computation, using the GETCOND command format, with a constant that is compared to the flag in each tuple. If these are equal, the property part of the tuple is sent to the next step (Normalizer), otherwise it is dropped.

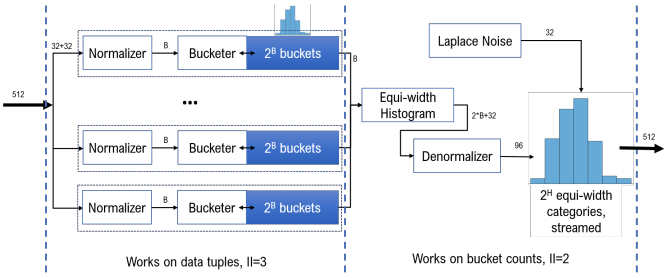

Fig. 3. The histogram computation is performed in two phases. First, tuples are counted into high resolution buckets by several parallel units. Second, these array data structures are merged and the final, course granularity, histogram is created ($H < B$).

The Normalizer step converts 32bit unsigned integers to B-bit unsigned integers using Min-Max normalization. This is done so that the Bucketer module can create a high resolution histogram with 2^B buckets of the data. In our prototype B=12 but B=16 could also be a reasonable value to pick – of course, with a larger value of B the cost of computing the final equi-width histogram grows linearly.

Overall, the performance bottleneck of this design is in this Normalize+Bucketer step, which can only process data with an initiation interval (II)=3. Therefore, in order to handle one 512bit input per cycle, 24 parallel units are instantiated in our design. The second part of the computation (right-hand side of Figure 3) has an II=2 but it works on smaller data than the input (2^B bytes instead of potentially megabytes). Also, it is very difficult to parallelize that step given the need to construct one merged data structure.

Partial counts are merged after all input has been consumed and each parallel bucketer. The merging step results in an Equi-Width Histogram with 2^H buckets. H can be configured at run-time and, to provide compact statistics to clients, H will typically be at least an order of magnitude smaller than B. At the end of this histogram creation step, a stream of {`bucket_begin, bucket_end, count`} tuples are sent to the Denormalizer. The Denormalizer performs the reverse of the initial Min-Max normalization on the bucket boundaries, bringing this way the histogram into the same domain as the original input. The count of each bucket is not modified by this step but noise is added to them as the last operation. This is done in a streaming fashion and the result is packed onto a 512bit wide AXI Stream.

For space reasons, we omit the details of metadata handling that is necessary for Multes to send responses back to clients and have to be propagated through the histogram module.

B. Noise Generation

To add noise to DP results, a Laplace distribution needs to be sampled. In our implementation, we approximate this by using a uniform pseudo-random number generator of 32bit size, based on a well-established cellular automata design [12], and adding by transforming these to values from a Laplace distribution. Having a random variable X drawn from a uniform random distribution with values in the $(-\frac{1}{2}, \frac{1}{2})$ interval, the random variable L will have a Laplace distribution with scale b, according to the following formula: $L = -b * sgn(X) * ln(1 - 2|X|)$. Thanks to the fact that the C++ cmath library works with HLS, we were able to implement this step without having to make further changes apart from re-mapping the random number output from $[0, 2^{32})$ to $(-\frac{1}{2}, \frac{1}{2})$. As will be seen in the Resource Consumption, this design requires several DSPs but the base Multes system uses almost none, so it is a good tradeoff.

When configuring the noise generator, the magnitude of the noise depends, among other factors on the "privacy budget" allocated. We utilize a Laplace distribution of scale $1/\epsilon$, as in related work [14]. For instance, when $\epsilon = 1$, our noise generator will add up to ± 2 to the count of each bucket. The choice of ϵ depends on various security factors and, thus, we allow the scale of the noise to be configured at run-time, instead of hard-coding it.

The random number generator is not in the critical path of performance. Nonetheless, the module is able to generate 32bit outputs with II=1, more than sufficient for the final histogram, which creates its buckets with only II=3.

C. Read Rate Limiter

Generating histograms repeatedly on the same dataset will, over time, reduce the privacy guarantees each individual entry benefits from. For this reason, it must be possible to limit the rate at which data can be used for histogram creation by unprivileged users. We investigated two ways of doing so.

The first option relies on the existing rate limiters of Multes. By modifying them to account for the data sizes retrieved from storage, instead of those sent/received over the network, it is possible to ensure that the underlying data is not read at a rate higher than what the privacy budget allows. This approach, however, is conservative because it depletes the global privacy budget regardless of the locality in accesses.

The second option, which we use in our prototype, adds a "last read" metadata field to all key-value entries. All key-value accesses with the purpose of creating a histogram (i.e., GETCOND operation) will be checked against the budget criteria. Overall, this approach gives a more fine-grained control but requires an additional hash table write for each GETCOND operation. In our prototype this is not an issue because HBM used for storing the hash table is fast enough.

IV. Evaluation

We evaluate our prototype using a Xilinx Alveo U280 card plugged into a server machine with an AMD EPYC 7402P processor and 128GBs of DRAM. The FPGA and the server communicate through a 100Gbps link. We have configured Multes to use HBM memory for its data storage and relied on the networking-enabled XACC shell for Alveo to provide 100Gbps TCP/IP connectivity. Due to the limitation of only one machine, we conducted measurements directly on the FPGA, using debugger probes to measure processing rates.

Q1: Does the additional bookkeeping inside the hash table reduce performance? We compared the performance of Multes

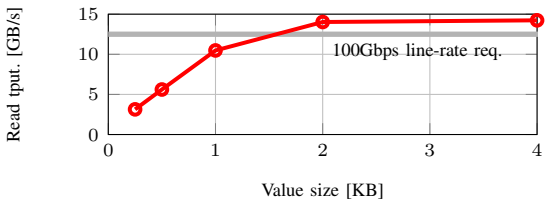

Fig. 4. The internal throughput of the Multes pipeline can match 100Gbps line-rate for GETs when the average value size is above 1.5KB. Below this number, the hash table logic is the bottleneck. The presence of DP rate-limiting logic has no measurable impact on performance (when not throttling).

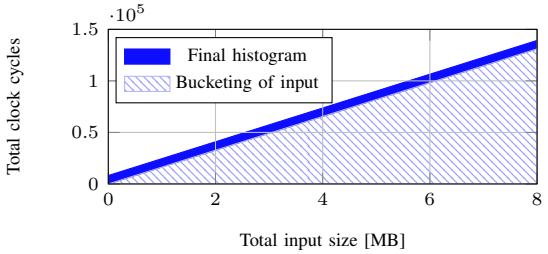

Fig. 5. Using the Histogram module for small datasets will introduce significant latency overhead, due to the final step of the computation (buckets-to-histogram with noise). For datasets of at east a few MBs, the share of this overhead becomes negligible.

with our prototype implementation and observed equal behavior. We plot the internal bandwidth of the key-value store in Figure 4, as a function of request size. The figure contains one line because the performance of the two variants is identical because the HBM on the FPGA is fast enough to handle one additional write operation on each access. Once the values are larger than 1.5KB, 100Gbps line-rate can be achieved. The histogram functionality does not modify writes or deletes so we omit such experiments for the sake of succinctness.

Q2: What is the cost of computing a histogram? The cost consists of two parts: i) processing the data in several key-value pairs into an intermediary representation (buckets) and ii) creating the final histogram with noise based on the buckets. The first cost is a direct function of the amount of data sent to the histogram module, which runs at constant 512 B/s speed at 250MHz. The second step is determined by the bucket count (4096 in our case) because it processes one bucket every two cycles. Figure 5 shows that, not surprisingly, for small datasets (less than 2MB) the constant cost of histogram creation will be a significant overhead, for larger datasets, this shrinks to a less than 10% overhead.

Q3: How many additional resources are used? The overhead, in terms of resources, of the rate limiting module is negligible. The overhead of the histogram module, especially the parallel bucketing steps is, however, significant. In Figure 6 we show the resources used by i) the Vitis shell on the U280, ii) by Multes with rate limiting and iii) by the histogram module and noise generator. Due to the 24 parallel units used for bucketing and the wide, 512 b input streams, the design requires a significant number of BRAMs for this part. The number of BRAMs increases linearly with the number of

Component (Available on device)	**LUT** 1303k	**BRAM** 4032	**URAM** 960	**DSP** 9024)
Platform (shell)	253k	632	19	4
Multes and Rate Limit	42k	280	24	0
Histogram and Noise	101k	220	0	211

Fig. 6. As expected, the histogram module requires a high number of BRAMs. Their number scales linearly with the parallelism level of the bucketers and the resolution of the intermediary data structure (default 4096 buckets).

parallel pipelines. The HLS compiler is able to utilize DSPs for the normalization operations, which is good because no other modules use DSPs.

V. Conclusion

This work demonstrates the feasibility of implementing histogram computation with differential privacy inside a Smart Storage node. This has been motivated by a larger effort of adding more privacy-preserving computation to the storage layer of data-intensive analytics applications. The implemented prototype modifies an open-source key-value store on a 100Gbps Alveo U280 board. As the Evaluation shows, all steps required can be implemented on the FPGA and the resource consumption of the new modules scales linearly with their nominal processing rate.

Acknowledgment

We would like to thank Xilinx for their generous donation of hardware and software used in this work.

References

[1] AQUA (advanced query accelerator) for amazon redshift. accessed 01/06/21. https://aws.amazon.com/redshift/features/aqua/.

[2] S. Bavadekar, A. Dai, J. Davis, D. Desfontaines, I. Eckstein, et al. Google covid-19 search trends symptoms dataset: Anonymization process description (version 1.0). *arXiv preprint arXiv:2009.01265*, 2020.

[3] J. Do, Y.-S. Kee, J. M. Patel, et al. Query processing on smart SSDs: opportunities and challenges. In *SIGMOD'13*.

[4] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, 2006.

[5] Z. István, S. Ponnapalli, and V. Chidambaram. Software-defined data protection: low overhead policy compliance at the storage layer is within reach! *Proceedings of the VLDB Endowment*, 14(7), 2021.

[6] Z. Istvan, L. Woods, and G. Alonso. Histograms as a side effect of data movement for big data. In *SIGMOD'14*.

[7] S.-W. Jun, M. Liu, S. Lee, J. Hicks, et al. Bluedbm: Distributed flash storage for big data analytics. *ACM Transactions on Computer Systems (TOCS)*, 34(3), 2016.

[8] P. Mehra. Samsung smartssd: Accelerating data-rich applications. *Proceedings of the Flash Memory Summit*, 2019.

[9] B. Salami, G. A. Malazgirt, O. Arcas-Abella, A. Yurdakul, and N. Sonmez. Axledb: A novel programmable query processing platform on fpga. *Microprocessors and Microsystems*, 51, 2017.

[10] A. Shahbahrami, J. Y. Hur, B. Juurlink, and S. Wong. Fpga implementation of parallel histogram computation. In *HiPEAC Workshop on Reconfigurable Computing*, 2008.

[11] S. Shastri, M. Wasserman, and V. Chidambaram. The seven sins of personal-data processing systems under GDPR. In *USENIX HotCloud'19*, 2019.

[12] D. B. Thomas and W. Luk. High quality uniform random number generation using lut optimised state-transition matrices. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 47(1):77–92, 2007.

[13] R. J. Wilson, C. Y. Zhang, W. Lam, et al. Differentially private sql with bounded user contribution, 2019.

[14] J. Xu, Z. Zhang, X. Xiao, Y. Yang, G. Yu, and M. Winslett. Differentially private histogram publication. *The VLDB Journal*, 22(6), 2013.